.NET 精选系列

SAMS

21天学通C#

〔美〕Bradley L. Jones 著
信达工作室 译

Microsoft .net

美术编辑：胡平利

21天学通C#

ISBN 7-115-10199-X
9 787115 101990 >

ISBN7-115-10199-X/TP·2831
定价：60.00 元

人民邮电出版社

.NET 精选系列

21 天学通 C#

[美] Bradley L. Jones 著

信达工作室 译

人民邮电出版社

图书在版编目（CIP）数据

21 天学通 C#/（美）琼斯（Jones, B.L.）著；信达工作室译.
—北京：人民邮电出版社，2002.3
（.NET 精选系列）
ISBN 7-115-10199-X

I. 2… II. ①琼… ②信… III. 语言—程序设计
IV. TP312

中国版本图书馆 CIP 数据核字（2002）第 009826 号

.NET 精选系列
21 天学通 C#

◆ 著　[美] Bradley L.Jones
译　信达工作室
责任编辑　李　际

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
读者热线　010-67180876
北京汉魂图文设计有限公司制作
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092　1/16
印张：36.5
字数：963 千字　2002 年 3 月第 1 版
印数：1-4 000 册　2002 年 3 月北京第 1 次印刷

著作权合同登记　图字：01－2001－4093 号
ISBN 7-115-10199-X/TP・2831
定价：60.00 元

作者简介

Bradley L. Jones 是包括 CodeGuru.com、Developer.com 和 Javascripts.com 等在内的众多著名开发人员网站的系统管理员，还是 internet.com 网站的 EarthWeb 频道的执行编辑。Bradley 使用 C#的时间比大多数开发人员都长，因为他被邀请在 beta 版发布之前对其进行试用。Bradley 拥有使用 C、C++、PowerBuilder、SQL Server 以及许多其他工具和技术进行开发工作的经验。另外，它还是一位畅销书作者，编写了最早的“21 天”图书——《21 天学通 C 语言》。

前 言

欢迎使用本书，通过本书的名称，您可能猜到了，作者编写本书旨在让您用 21 天的时间学通编程语言 C#。本书分为 21 天课程，其中每天的课程都能在几个小时（或一个晚上）完成。读者在 21 天中每天都花 2～3 个小时，则将轻松地学习完本书。当然，这不一定都是连续的 21 个晚上，也不一定非得在晚上阅读。

其中每天课程都可以在 1～2 小时内阅读完毕，有些可能所需的时间长些，而有些可能短些。如果读者期望仅通过阅读便能学通 C#，将肯定会失望。相反，您应该将一半的时间用于阅读，另一半的时间用于输入其中的代码、完成小测验和练习。这看起来工作量好像很大，但每天课程都可以在一个晚上完成。

其中每天课程都包括小测验和练习，它们是为帮助您巩固对当天内容的理解而设计的。阅读完每天课程的内容后，读者应该知道所有的小测验答案，如果不是这样，则应该复习该课中的一些内容。练习给读者提供了应用所学知识的机会。通常，练习的重点在于理解代码、识别常见的代码错误或根据当天的课程编写代码。

小测验和大部分练习的答案都可以在附录 A 中找到，在参考其中的答案之前，请自己尽可能地解决。

阅读本书时，您还将发现其几个特点。书中包含“提示”、“注意”和“警告”。“提示”提供了很有帮助的建议；“注意”提供了一些您可能感兴趣的信息；而“警告”则提醒您注意一些可能遇到的问题。本书中一个很特别的部分是每一课程最后的“问与答”一节，其中包含您在阅读该课程时可能提出的问题及其答案。这些问题可能涉及到与该课程相关的主题。

对读者的要求

作者对读者做了一些假设。假设读者有 C#编译器和.NET 运行阶段环境。即使没有这些东西，读者也能阅读本书，但要完全理解其中

的内容将很困难。

作者假设读者是初中级程序员，如果您不是，也将从本书中学到很多东西，但在某些地方，您的进展速度将没有您希望得那么快。

本书不要求读者使用 Visual C#或 Visual Studio.NET 开发环境。许多图书要求读者使用微软公司的这些工具，但本书不要求。读者可以使用微软公司的工具，也可以使用许多其他的工具。阅读本书时，您将对此有更深刻的体会。

网站支持

没有人是十全十美的——特别是作者。加上本书介绍的是一种较新的编程语言，因此您可能会发现书中的一些问题。

本书已经多人审校，除了作者本人的校对外，还有一些人对其做了技术和开发方面的审阅。虽然经过这么多的审阅，但错误仍将难免。为防出现问题，多个网站都提供了本书的勘误表，其中出版社网站的网址为：http://www.samspublishing.com/；另外作者也为本书专门创建了一个网站，其网址为 www.TeachYourseltCSharp.com，我将在这里发布勘误表。

源代码

作者认为，学习编程语言的最佳方式是输入代码并运行之。为此，笔者不想在初中级图书中附带光盘。但我也知道，并非每个人都认同这种观点，因此我在网站 www.TeachYourseltCSharp.com 上提供了本书的源代码。

本书是为学习编写的，读者可以使用其中的源代码，对其进行修改、扩展，甚至提供给他人。购买本书后，您便有权以任何您认为合适的方式使用其中的代码，只有一种情况例外，那就是不能将其用于其他的 C#教学材料中。

作　者

目 录

第一周课程

第一周课程

欢迎使用本书。如果您还不知道阅读本书大部分内容所需的知识，请参阅前言，其中还介绍了本书的内容。

现在开始阅读第 1 周的课程，它将帮助您牢固地掌握编写 C#程序的基本知识。第 1 天的课程“C#初步”将带您进入 C#程序，并介绍创建 C#程序的步骤。

第 2 天课程“了解 C#程序”介绍 C#是如何适应 Microsoft .NET 框架的，还将介绍面向对象编程的基本原理。

第 3 天课程“使用变量存储信息”、第 4 天课程“使用运算符”、第 5 天课程“控制语句”将介绍 C#编程中必备的核心编程概念，其中包括存储和操纵数据以及控制程序流程方面的概念。

第 6 天课程“类”和第 7 天课程“类方法和成员函数”介绍了类和类方法。类是面向对象编程的核心概念，因此也是 C#编程的核心概念。

阅读完本周的课程后，您将学到 C#编程的许多基础概念。您将发现，您已掌握了构建基本 C#程序的工具和知识。

第 1 天课程

C#初步

欢迎阅读本书，今天将开始成为熟练的 C#程序员之旅。今天将介绍以下内容：

- 为什么说 C#是一种非常棒的编程语言；
- 程序开发周期中的各个步骤；
- 编写、编译和运行第 1 个 C#程序；
- 编译器和链接程序生成的错误消息；
- 可以使用 C#来创建的应用程序类型。

1.1 C#是什么

如果没有听说过 C#，则您购买本书的可能性不大，但很可能对其了解不深。C#（读作 see Sharp）是在 2000 年 6 月发布的，已经面世了很长时间。

C#是微软公司开发的一种新语言，并已提交给 ECMA 进行标准化。该新语言是由微软公司的一个小组开发的，带头人是 Anders Hejlsberg。有趣的是，Hejlsberg 是微软公司杰出的工程师，曾经开发了其他一些产品和语言，其中包括 Borland Turbo C++和 Borland Delphi。开发 C#时，Hejlsberg 着重借鉴现有语言中的优点，并做一些改进。

C#是一种功能强大的、灵活的编程语言，和所有的编程语言一样，它也可用于创建各种应用程序。C#的用途只受您的想象力的限制，该语言没有给您任何约束。C#已被用于创建动态网站、开发工具，甚至编译器。

C#是一种面向对象的编程语言（OOP）。其他编程语言也包含面向对象的特性，但很少是完全面向对象的。本章后面将对 C#和其他一些编程语言进行比较，同时介绍可用它来创建的应用程序类型。在第 2 天的课程“了解 C#程序”中，您将知道使用面向对象的编程语言意味着什么。

1.2 为何使用 C#

很多人认为，根本不需要一种新的编程语言，他们认为 Java、C++、Perl、Microsoft Visual Basic 以及其他的一些语言已提供了所需的所有功能。

C#是一种从 C 和 C++派生而来的语言，但却是完全重新开发的。微软公司借用了 C 和 C++中一些已有的东西，并加入了新特性，这些特性使该语言更容易使用。其中的许多特性与 Java 极其类

似。最终，微软公司在开发该语言时，有很多目标，其中一些包含在微软公司关于 C#的声明中：

- C#简单；
- C#现代；
- C#是面向对象的。

除了上述原因外，还有其他一些原因：

- C#强大、灵活；
- C#语言包含的单词为数不多；
- C#是模块化的；
- C#将流行。

警告：下一节包含大量的技术术语，但不用担心。其中大部分术语对C#程序员而言，是无关紧要的！对于那些重要的术语，本书后面将对其进行解释。

1.2.1　C#简单

C#消除了诸如 Java 和 C++等语言中复杂的东西和缺陷，其中包括宏、模板、多重继承和可视化基类。这些内容可能让 C++开发人员困惑或带来问题。如果 C#是您学习的第 1 种语言，则请放心，您不必花时间来学习这些主题。

C#之所以简单是由于它是基于 C 和 C++的。如果您熟悉 C 和 C++（甚至 Java），则您将发现 C#在很多方面与它们极其类似。语句、表达式、操作符以及其他一些功能是直接来自 C 和 C++的，但做了一些改进，使该语言更为简单。其中包括消除了冗余的东西以及语法上的变化。例如，在 C++中使用成员时，可用的操作符有三个："::"、"." 和 "->"。在何时使用其中的某一个操作符是令人迷惑的。在 C#中，这些操作符被一个操作符 "." 所代替。对于新程序员而言，这种特性（以及其他一些特性）避免了许多困惑。

注意：如果您使用过Java，并认为它简单，则您将发现C#也简单；大多数人并不认为Java简单，但却认为C#比Java和C++都简单。

1.2.2　C#现代

什么因素决定一种语言为现代语言呢？一种现代语言必须包含诸如异常处理、无用单元收集、可扩展数据类型以及代码安全性等特性。C#具备上述所有特性。如果您是一名新程序员，则可能想知道这些看起来很复杂的特性，阅读本书后，您将知道它们非常适用于 C#编程技术！

注意：指针是C和C++的有机组成部分，也是这些语言中最令人困惑的地方。C#消除了指针导致的复杂性和麻烦，在C#中，自动无用单元收集和类型安全是其有机组成部分。如果您不熟悉指针、无用单元收集和类型安全的概念，请不要担心，本章后面将介绍这些概念。

1.2.3　C#是面向对象的

新术语：对于面向对象的语言而言，最重要的是封装、继承和多态，而C#支持所有这些特性。*封装*指的是将功能放在单个组件中；*继承*是一种将已有的代码和功能扩展到新的程序和组件中的结构化方式；*多态*指的是适应所需完成的工作的能力。关于这些术语和面向对象的更详细的解释，请参考第2天的课程。另外，本书将更为详细地讨论这些主题。

1.2.4　C#强大、灵活

正如前面指出的，使用 C#您只受想象力的限制，该语言对您没有任何约束。C#可用于创建字

处理程序、图形、电子表格，甚至其他语言的编译器。

1.2.5 C#包含的单词为数不多

新术语：C#只使用有限的几个单词。语言的功能是建立在关键字的基础之上的，C#包含的关键字不多。表1.1列出了C#使用的关键字，其中的大部分用于描述信息。您可能认为，包含的关键字越多，语言的功能越强大，其实并非如此。当您使用C#编写程序时，将发现它可用于完成任何任务。

表 1.1　　C#中的关键字

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
while				

注意：C#还使用了其他几个单词，虽然它们不是关键字，但应该将其看作是关键字。具体地说，它们是get、set和value。

1.2.6 C#是模块化的

新术语：C#代码可以（也应该）以程序块——被称为类的方式编写，类包含被称为成员方法的例程。这些类和方法可以在其他应用程序中被重用。通过给类和方法传递信息，可以创建出很有用的、可重用的代码。

注意：第2天的课程将介绍类，而第6天的课程将介绍如何创建类。

1.2.7 C#将流行

C#是最新的编程语言之一，编写本书时，对 C#的流行程度还不得而知，但基于很多原因，它

肯定会成为一种非常流行的语言。其中最关键的原因之一是微软公司及其.NET 的远大前程。

微软公司希望 C#流行。虽然一个公司无法使其产品流行，但可以辅助其流行。不久前，微软公司在其操作系统 Bob 上遭受了重大的失败，虽然微软公司期望 Bob 流行，但失败了。

C#比 Microsoft Bob 成功的可能性更大，我不知道微软公司的员工在其日常工作中是否使用 Bob，但微软公司确实正在使用 C#，其许多产品中的一些代码已经使用 C#重新编写过。通过使用 C#，微软公司帮助证明了 C#确实能够满足程序员的需求。

Microsoft .NET 是 C#将成功的另一个原因。.NET 改变了创建和实现应用程序的方式，在.NET 中，几乎可以使用任何编程语言，C#被证明是可供选择的语言。明天的课程中包含介绍.NET 要点的一节。

C#还将因前面提到的所有特性而流行：简单性、面向对象、模块性、灵活性以及简明性。

1.3　C#和其他编程语言的比较

您可能听说过 Visual Basic、C++和 Java。您可能会问，C#和这些编程语言之间有何差别呢？是否应该学习上述三种语言之一，而不是 C#呢？

注意：在Internet上，与.NET相关的论坛讨论得最多的问题如下：

- Java 和 C#之间的区别何在？
- C#是否只是 Java 的克隆而已？
- C#和 C++的区别何在？
- 应该学习 Visual Basic.NET 还是 C#？

微软公司宣称，C#融 C++的强大和 Visual Basic 的简易性于一体。C#确实很强大，但和 Visual Basic 一样简单吗？它可能没有 Visual Basic 6 那么简单，但和 Visual Basic.NET（第 7 版）一样简单，Visual Basic.NET 是重新编写的。因此最终结果是，Visual Basic 确实不比 C#简单。事实上，无论编写任何程序，使用 C#所需的代码都将更少。

虽然 C#删除了 C++让程序员痛苦的一些特性，但其强大性或功能并没有降低。对于 C++很容易出现的编程错误，在 C#中已经可以完全避免了。这在开发程序的时间上，可以节省几个小时，甚至几天。学习本书介绍的主题后，您将对 C#和 C++之间的差别有更深入的了解。

另一种引起了人们很多关注的语言是 Java，与 C++和 C#一样，Java 也是基于 C 语言的。如果您决定以后学习 Java，您将发现在 C#中学到的许多知识也适用于 Java。

您可能还听说过 C 语言。很多人有这样的疑问，即在学习 C#、C++或 Java 之前，是否应学习 C 语言。简单地回答，这完全没有必要。

知道足够多的原因和结论后，您很可能购买本书，以便学习使用 C#来编写程序。接下来的几节将介绍开发程序的步骤，然后从头到尾开发一个简单的程序。

1.4　编程前的准备工作

您解决某个问题时，总是需要采取一些特定的步骤。首先必须确定问题。如果不知道问题是什么，则不可能找到解决之道！知道问题后，便可以制定解决问题的方案。有了方案，通常可以实现之。方案实现后，必须对结果进行测试，以便了解问题是否得到了解决。这种逻辑也适用于包括编程在内的其他许多领域。

使用 C#（或其他任何语言）创建程序时，应该按类似于下面的步骤进行：

1. 确定目标；
2. 确定编写程序时采用的方法；
3. 创建程序，以解决问题；
4. 运行程序，以查看结果。

目标（见第 1 步）可能是编写一个字处理器或数据库程序。一个极其简单的目标是将您的姓名显示在屏幕上。没有目标，则无法编写出有效的程序。

第 2 步是确定编写程序所采用的方法。是否需要一个计算机程序来解决问题？需要记录哪些信息？要使用哪些公式？这一步应确定需要什么以及解决方案的实现次序。

作为一个例子，假设有人叫您编写一个计算圆面积的程序，则第 1 步已经完成，因为您知道目标：计算圆的面积。第 2 步是确定为了计算面积需要什么。这里假设程序用户将提供圆的半径。知道这一点后，您便可以使用公式 πr^2 来获得答案。有了所需的信息后，便可以接着完成第 3 步和第 4 步，这两步被称为程序开发周期。

1.5 程序开发周期

程序开发周期有其自己的步骤。第 1 步是使用编辑器创建包含源代码的文件；第 2 步是编译源代码，以创建被称为可执行文件或库文件的中间文件；第 3 步是运行程序，以查看它是否按原来预想的情况运行。

1.5.1 创建源代码

新术语：*源代码*是一系列指示计算机执行要完成的任务的语句和命令。前面指出过，程序开发周期的第 1 步是在一个编辑器中输入源代码。例如下面是一行 C#源代码：

```
System.Console.WriteLine("Hello, Mom!");
```

该语句命令计算机在屏幕上显示消息 "Hello, Mom!"（现在，请不用担心该语句是如何工作的）。

1.5.1.1 使用编辑器

新术语：*编辑器*是一个可用于输入和保存源代码的程序，对于 C#，可用的编辑器很多，其中的一些是专门为 C#开发的，而其他的不是。

编写本书时，为 C#开发的编辑器为数不多，但随着时间的推移，将有更多的编辑器面世。微软公司在其 Visual Studio 产品中添加了 C#功能，该产品包含 Visual C#。这是最主要的编辑器。但即使没有 Visual Studio.NET，仍然可以使用 C#进行编程。

还有其他可用于 C#的编辑器。和 Visual Studio.NET 一样，其中的很多让您不用离开编辑器便可以完成开发周期的所有步骤。更重要的是，这些编辑器大部分都将使用不同的颜色来显示您输入的文本，这使得查找错误的工作更为容易。许多编辑器甚至会提供关于您需要输入什么内容的帮助信息，并提供了一个健壮的帮助系统。

如果没有 C#编辑器，也不用烦恼。大多数计算机系统都包含一个可用作编辑器的程序。如果您使用的操作系统是 Microsoft Windows，则可以将 NotePad 或 WordPad 用作编辑器；如果使用的是 Linux 或 UNIX 系统，则可以使用诸如 ed、ex、edit、emacs 或 vi 等编辑器。

大多数字处理程序都使用特殊的代码来格式化其文档，其他程序无法正确地读取这些代码。许

多字处理程序（如 WordPerfect、Microsoft Word 和 WordPad）能够将源文件保存为文本格式。要将字处理程序的文件保存为文本文件，请在保存时选择文本选项。

注意：可以到计算机商店、计算机邮购目录和计算机编程杂志的广告中寻找其他的编辑器。编写本书时，下面几个编辑器已经面市：

- CodeWrite：一个为ASP、XML、HTML、C#、Perl、Python等提供特殊支持的编辑器，可以从www.premia.com下载。
- EditPlus：一种Internet-就绪（ready）的文本编辑器、HTML编辑器和Windows环境下的程序员编辑器。可以替代NotePad，同时为Web页开发人员和程序员提供了许多强大的特性，包括使用不同的颜色显示代码。该编辑器可以从www.editplus.com下载。
- JEdit：用于Java的开发源代码（Open-Source）编辑器，也可用于C#，能够以不同的颜色显示代码。该编辑器可以从http://jedit.sourceforge.net下载。
- Duncan Chen开发的Poorman IDE：提供了一个突显C#和Visual Basic.NET语法的编辑器，同时让您能够运行编译器和捕获控制台输出，因此您无需离开Poorman IDE。Poorman可以从www.geocities.com/duncanchen/poormanide.htm下载。
- Mike Krüger开发的SharpDevelop：是一个免费的Microsoft .NET平台上的C#编辑器。它是一个开放源代码编辑器，因此您可以从www.icsharpcode.net下载其源代码和执行文件。

1.5.1.2　给源代码文件命名

保存源代码文件时，应该提供一个描述其功能的名称。另外，对于 C#程序的源代码文件，应将其扩展名设置为.cs。虽然您可以给源代码文件指定任何的文件名和扩展名，但.cs 是公认的合适的扩展名。

1.5.2　执行 C#程序

在深入介绍程序开发周期之前，需进一步了解 C#程序是如何运行的，这非常重要。C#程序不同于使用其他编程语言创建的程序。

新术语：C#程序将在通用语言运行阶段环境（Common Language Runtime，或称 CLR）中运行。这意味着如果您创建一个 C#语言的可执行程序，并试图在一个没有安装 CLR 或兼容运行阶段环境的机器上运行它，则将无法执行。*可执行*意味着程序可以被计算机运行或执行。

创建运行阶段环境程序的好处在于可移植性。在诸如 C 和 C++等以前的语言中，要创建一个可在不同的平台（操作系统）上运行的程序，必须编译不同的可执行程序。例如，编写 C 应用程序时，如果希望它能够在 Linux 和 Windows 系统上运行，则必须创建两个可执行程序——一个用于 Linux 机器，另一个用于 Windows 机器。使用 C#时，只需要创建一个可执行程序，该程序可在上述任何机器上运行。

新术语：如果希望程序的执行速度尽可能快，则需要创建一个真正的可执行程序。计算机使用的是数字（二进制）指令，这被称为*机器语言*。程序必须从源代码被翻译为机器语言，这项工作是由一种叫作编译器的程序完成的。编译器利用源代码生成一个磁盘文件，该文件中包含与源代码语句对应的机器语言指令。对于诸如 C 和 C++等程序，编译器将创建一个无需做进一步的工作便可被执行的文件。

对于 C#，编译器并不生成机器语言文件，而是生成一个中间语言（Intermediate Language，或称 IL）文件。由于这种文件不能被计算机直接执行，因此需要对其进一步翻译或编译，以便计算机能直接执行它，这项工作是由 CLR 或其他兼容的 C#运行阶段环境完成的。

CLR 首先对 IL 文件做最终的编译，即将可移植的 IL 代码转换为计算机能够理解并运行的语言（机器语言）。CLR 实际上只编译程序中将被使用的部分，这样可以节省时间。另外，当 IL 文件中的某部分被转换为汇编语言后，便无需再次进行编译，因为编译后的代码将被保存，以后该部分被执行时，将直接使用编译后的代码。

注意：由于运行阶段环境需要编译IL文件，因此与诸如C++等编译型语言相比，程序首次的运行时间可能稍长。程序首次被完全执行后，这种时间上的差别将不复存在，因为从此以后被执行的将是完全被编译的版本。

注意：C#程序在最后一刻编译的特性被称为即时编译（Just In Time compiling，或称jitting）。

1.5.3 编译 C#源代码

要创建 IL 文件，需要使用 C#编译器。通常，您使用 csc 命令（后面带源代码文件的名称）来运行编译器。例如，要编译名叫 radius.cs 的源代码文件，可以输入下述命令行：

```
csc radius.cs
```

如果您使用的是图形开发环境，则编译工作将更为简单。在大多数图形环境中，您都可以通过单击“编译”工具栏按钮或选择合适的菜单选项来编译程序。代码被编译后，可以单击工具栏按钮“运行”或选择合适的菜单选项来执行程序。有关编译和运行程序的具体方法，请参阅编译器的手册。

新术语：编译后，将得到一个IL文件。如果您查看对应目录中的文件清单，将发现一个扩展名为.exe的新文件，其文件名与源代码文件相同。该文件是编译后的程序，称为*组合体*（assembly），能够在CLR上运行。组合体文件包含CLR为执行程序所需的所有信息。

图 1.1 说明了从源代码到可执行文件的过程。

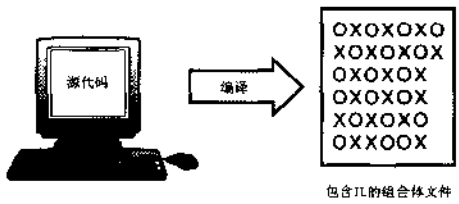


图 1.1 您编写的 C#源代码被编译器转换为中间语言（IL）代码

注意：通常有两种可交付使用的C#程序：可执行程序和库。本书前两周的课程重点介绍可执行文件，其扩展名为.exe。也可以将C#用于另一种编程，即编写ASP.NET页面上的脚本。有关库的知识将在第三周的课程中介绍。

1.5.4 完成开发周期

程序被编译为 IL 文件后，便可以在命令行提示符下输入其名称来运行它（就像运行其他任何程序一样）。

运行程序时，如果得到的结果与原来预想的不同，则需要回到开发周期的第一步，确定导致问题的原因，并在源代码中进行更正。修改源代码后，需要对其重新进行编译，以创建更正后的可执行文件版本。这一过程将一直重复下去，直到程序的执行情况与预想的完全一致。

C#开发周期

第 1 步：使用编辑器编写源代码。C#源代码文件的扩展名通常为.cs（如 a_program.cs、database.cs 等）；

第 2 步：使用 C#编译器对程序进行编译。如果编译器发现程序没有任何错误，则将生成一个扩展名为.exe 或.dll 的组合体文件。例如，默认情况下，myprog.cs 被编译为 myprog.exe；如果发现错误，编译器将报告。在这种情况下，您必须返回到第 1 步，对源代码进行更正。

第 3 步：在安装了 C#运行阶段环境（如 CLR）的机器上执行程序。此时，应对程序进行测试，以确定它是否正确地运行。如果不是，则返回到第 1 步，对源代码进行修改。

图 1.2 说明了程序开发的步骤。除了最简单的程序外，您可能都得重复这一过程多次才能完成程序。即使是最有经验的程序员，也不可能一步就编写出一个完整的、没有错误的程序！由于需要重复编辑—编译—测试多次，因此熟练使用工具——编辑器、编译器和运行阶段环境——至关重要。

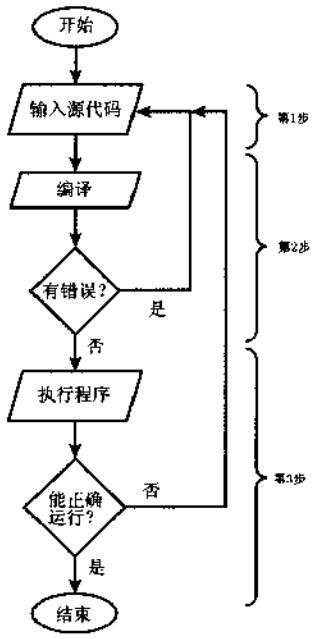


图 1.2　开发 C#程序的步骤

1.6　第 1 个 C#程序

您可能迫切地希望编写第 1 个 C#程序。为帮助您熟悉编译器，程序清单 1.1 包含一个您很快就能完成的程序。当前，您可能不明白其中的所有内容，但可以体会一下编写、编译和运行真正的 C#程序的过程。

这里使用的是一个名为 hello.cs 的程序，它只是将单词“Hello, World!”显示到屏幕上。该程序常被人们用来介绍如何编程，对于学习 C#，这也是一个很好的例子。hello.cs 的源代码如程序清单 1.1 所示。输入该程序清单时，请不要输入左边的编号。

程序清单 1.1　hello.cs

```
1: class Hello
2: {
```

```
3:    static void Main()
4:    {
5:       System.Console.WriteLine("Helio, World!");
6:    }
7: }
```

请务必按软件安装提示安装编译器。准备好编译器和编辑器后，请按下一节中介绍的步骤输入、编译和执行 hello.cs。

1.6.1 输入并编译 hello.cs

请按下述步骤输入并编译 hello.cs：

1. 启动编译器。

2. 使用键盘输入如程序清单 1.1 所示的 hello.cs 中的源代码。请不要输入其中的行号，这些行号只是为方便引用而提供的。在每一行结束时按 Enter 键。输入代码时，大小写一定要相同。C#是区分大小写的，因此如果您改变了大小写，将发生错误。

警告： 如果您是C或C++程序员，则很可能犯一种常见的错误。在C和C++中，main()是小写的，而在C#中，Main()中的M是大写的。如果您输入的是小写的m，将出错。

3. 使用文件名 hello.cs 保存源代码。

4. 查看目录中的文件，确保 hello.cs 被保存了。

5. 编译 hello.cs，如果使用的是命令行编译器，则使用下面的命令：

```
csc hello.cs
```

如果使用的是集成开发环境，则单击合适的图标或选择合适的菜单项。此时应显示一条消息，指出没有错误或警告。

6. 查看编译器给出的消息。如果没有错误或警告，则万事大吉。

如果您输入程序时有错，则编译器将捕获这种错误，并显示错误消息。例如，如果将单词 Console 错误地拼写为 Consol，则您将看到与下面类似的消息：

```
hello.cs(5,7): error CS0117: 'System' does not contain a definition for 'Consol'
```

7. 如果出现错误消息，则回到第 2 步。在编辑器中打开文件 hello.cs，仔细地将文件的内容与程序清单 1.1 进行比较，修改错误的地方，然后进入到第 3 步。

8. 至此，您的第一个 C#程序编译好了，可以运行。如果显示文件名为 hello 的文件，则将看到如下两个文件：

- hello.cs：使用编辑器创建的源代码文件；
- hello.exe：编译 hello.cs 时生成的可执行程序。

9. 在命令行输入 hello，执行 hello.exe，屏幕上将显示 “Hello, World!”。

注意： 如果在Windows资源管理器中双击文件hello.exe，将可能看不到结果。该程序将在命令行窗口的方式运行，当您双击该文件时，将打开一个命令行窗口，运行该程序，当程序运行完毕后窗口将被关闭。这一过程持续的时间非常短，您可能没有看到任何情况发生。因此，最好打开一个命令行窗口，切换到包含该程序的目录，然后在命令行下运行该程序。

祝贺您！您通过输入、编译，便可以运行第一个 C#程序。得承认，hello.cs 是一个非常简单的程序，它没有完成任何有用的工作，但这只是开始。事实上，当前的大多数专家级程序员都是以这

种方式——通过编译一个“Hello World”程序——开始学习的。

1.6.1.1　编译错误

当编译器发现源代码的某些内容无法编译时，便会出现编译错误。拼写错误、排列错误以及其他一些错误都可能导致编译器无法进行编译。所幸的是，现代的编译器不仅仅停止编译，还会告诉您问题所在。这使得查找和更正源代码中的错误更为容易。

为证明这一点，可以故意在前面输入的 hello.cs 程序中引入一个错误。如果您完成了前面的步骤，则磁盘上将有 hello.cs 的一个拷贝。使用编辑器，将光标移到第 5 行的最后，并删除分号，得到的 hello.cs 如程序清单 1.2 所示。

程序清单 1.2　包含一个错误的 hello.cs

```
1: class Hello
2: {
3:    static void Main()
4:    {
5:       System.Console.WriteLine("Hello, World!")
6:    }
7: }
```

接下来，保存该文件。然后输入如下所示的编译器命令，以编译该文件：

```
csc hello.cs
```

由于其中有错，因此编译将无法完成。编译器将显示一条与下面类似的消息：

```
hello.cs(5,48): error CS1002: ; expected
```

该行消息由下述三部分组成：

- hello.cs：有错误的文件的名称；
- (5,48)：指出错误的位置——第 5 行的第 48 个字符处；
- error CS1002:;expected：对错误的描述。

该消息提供了大量的信息，指出当编译器编译到文件 hello.cs 的第 5 行的第 48 个字符时，期望找到一个分号，但没有。虽然编译器在检测和定位错误方面非常聪明，但它毕竟不是爱因斯坦。您必须使用 C#语言方面的知识，对编译器提供的消息进行解释，并确定所报告的错误的实际位置。通常，可以在编译器指出的行中找到错误，但如果没有找到，则很可能在前一行。刚开始查找错误时，可能很困难，但不久后便可以得心应手。

结束该主题之前，来看另一个编译错误的例子。在编辑器中打开 hello.cs，并做如下修改：

1. 在第 5 行的末尾加上分号；
2. 删除单词 Hello 前面的双引号。

保存并编译该文件。这次编译器将显示与下面类似的消息：

```
hello.cs(5,32): error CS1010: Newline in constant
```

该错误消息正确地指出了错误的位置：第 5 行的第 32 个字符，这正是要显示的文本的第一个双引号所在的位置，但未能正确地指出代码中少了一个双引号。这里编译器对问题做出了猜想，虽然猜想离实际情况不远，但并不正确。

提示：如果编译器报告了多个错误，而您只找到一个。更正该错误并重新编译文件后，您可能发现只需更正这个错误，程序编译时便没有任何错误。

1.6.1.2 逻辑错误

您还可能犯另一种错误：逻辑错误。逻辑错误无法依靠编译器来发现，而只能依赖您自己。即使程序中的 C#代码是完整无缺的，但仍然可能有错误。例如，假设您需要计算圆的面积，但使用的公式却是圆周率和半径乘积的两倍：

面积= 2πr

将上述公式输入到程序，并编译和执行该程序时，您将得到答案。C#程序的语法是正确的，但每次运行该程序得到的答案却是错误的。因为其中的逻辑不正确，该公式计算的是圆周长，而不是面积。您应该使用公式 πr^2。

无论编译器多么优秀，都无法发现逻辑错误。您必须通过仔细地核对代码并运行程序来找到这种错误。

1.7 C#程序的类型

在结束今天的课程之前，有必要介绍一下使用 C#可以创建哪些类型的应用程序。使用该语言可创建的程序类型很多：

- 控制台应用程序：控制台应用程序是从命令行运行的，本书创建的都是控制台应用程序，它们主要是基于字符（或文本）的，因此理解起来相对较为简单。
- Windows 应用程序：也可以创建充分利用 Microsoft Windows 提供的图形用户界面（GUI）的 Windows 应用程序。
- Web 服务：Web 服务指的是可以通过 Web 调用的例程。
- Web 表单/ASP.NET 应用程序：ASP.NET 应用程序是在 Web 服务器上执行的，生成动态 Web 页。

除了上述应用程序外，C#还可用于完成大量其他的工作，其中包括创建库、控件等。

1.8 总 结

在今天课程的开始，介绍了 C#的特征，包括功能强大、灵活、面向对象等。另外，还指出了 C#简单而现代。

今天探讨了编写 C#程序的各个步骤——被称为程序开发过程。阅读后面的内容之前，您应该牢固地掌握编辑—编译—测试循环。

在程序开发中，错误是不可避免的。C#编译器能够发现源代码中的错误，并显示错误消息，指出错误的特征和位置。使用这些信息，您可以更正源代码中的错误。但需要牢记的是，编译器并不总能够准确地指出错误的特征和位置。有时候，您需要使用 C#方面的知识来找到导致错误消息的真正原因。

1.9 问与答

问：C#程序能够在任何机器上运行吗？

10. 下述错误很可能位于哪一列?

```
my_prog.cs(35,6): error CS1010: Newline in constant
```

1.10.2 练习

1. 使用文本编辑器查看编译程序清单 1.1 时创建的.exe 文件。.exe 文件与源代码文件类似吗?(退出编辑器时,请别保存该文件)

2. 输入下述程序,并编译之(不要输入行号)。该程序有何功能?

```
 1: // circle.cs - Using variables and literals
 2: // This program calculates some circle stuff.
 3: //-----------------------------------------------
 4:
 5: using System;
 6:
 7: class variables
 8: {
 9:     public static void Main()
10:     {
11:         //Declare variables
12:
13:         int radius = 4;
14:         const double PI = 3.14159;
15:         double circum, area;
16:
17:         //Do calculations
18:
19:         area = PI * radius * radius;
20:         circum = 2 * PI * radius;
21:
22:         //Print the results
23:
24:         Console.WriteLine("Radius = {0}, PI = {1}", radius, PI );
25:         Console.WriteLine("The area is {0}", area);
26:         Console.WriteLine("The circumference is {0}", circum);
27:     }
28: }
```

3. 输入下述程序,并编译之(不要输入行号)。该程序有何功能?

```
1: class Aclass
2: {
3:    static void Main()
4:    {
5:       int x,y;
6:       for ( x = 0; x < 10; x++, System.Console.Write( "\n" ) )
7:          for ( y = 0; y < 10; y++ )
8:             System.Console.Write( "X" );
9:    }
```

```
10: }
```

4. 下述程序有问题，请将其输入到编辑器中，并编译之。哪些行引发错误消息？

```
1: class Hello
2: {
3:    static void Main()
4:    {
5:       System.Console.WriteLine(Keep Looking!);
6:       System.Console.WriteLine(You'll find it!);
7:    }
8: }
```

5. 将练习 3 中的程序作如下修改，并重新编译和执行该程序。现在，该程序完成的工作是什么？

```
8:             System.Console.Write( "{0}",  (char) 1 );
```

第 2 天课程
了解 C#程序

除了了解程序的基本组成部分外，您还需要了解 C#程序的结构。今天的课程将介绍以下内容：

- C#应用程序的组成部分；
- C#语句和表达式；
- 面向对象编程技术的特征；
- 封装、继承、多态和重用；
- 在程序中显示基本信息。

2.1 C#应用程序

今天的课程首先重点探讨一个 C#应用程序范例。通过程序清单 2.1，您将了解 C#应用程序的一些重要组成部分。

程序清单 2.1 app.cs：一个 C#应用程序范例

```
1:  // app.cs - A sample C# application
2:  // Don't worry about understanding everything in
3:  // this listing. You'll learn all about it later!
4:  //-------------------------------------------------
5:
6:  using System;
7:
8:  class sample
9:  {
10:     public static void Main()
11:     {
12:        //Declare variables
13:
14:        int radius = 4;
15         const double PI = 3.14159,
16.        double area;
17:
18:        //Do calculation
19:
```

```
20:         area = PI * radius * radius;
21:
22:         //Print the results
23:
24:         Console.WriteLine("Radius = {0}, PI = {1}", radius, PI );
25:         Console.WriteLine("The area is {0}", area);
26:     }
27: }
```

您应该将该程序清单输入到编辑器中，然后使用编译器来创建可执行程序。可以将该程序保存为 app.cs，然后在命令提示符下输入下面的命令来编译该程序：

```
csc app.cs
```

如果您使用的是可视化编辑器，则应该可以通过菜单项选择编译器。

警告：记住，输入上述程序清单时，不要输入其中的行号。行号是为方便讨论程序清单而加入的。

运行该程序，将得到如下所示的输出：

```
Radius = 4, PI = 3.14159
The area is 50.3344
```

正如您看到的，该程序清单的输出简单明了。首先显示了半径和圆周率的值，然后显示了根据上述值计算得到的圆面积。

接下来的几节，将介绍该程序的各个部分。请不要因为不能理解其中的所有内容而担心，后面的课程将更详细地讨论这些内容。接下来的几节旨在让您有一个初步的认识。

2.1.1　注释

程序清单 2.1 的开始四行是注释。注释用于在程序中输入将被编译器忽略的信息。为何要输入被编译器忽略的信息呢？原因很多。

注释常被用来提供关于程序清单的描述性信息，例如身份证明信息。另外，通过加入注释，可以指出期望程序清单完成的功能。即使只有您自己使用程序清单，加入关于程序的功能及其如何完成这些功能的信息也是一个好主意。虽然您现在知道程序清单的功能——因为您刚编写它，但以后您可能不知道当时您是如何想的。当您将程序清单提供给他人时，注释将帮助他们了解代码要做什么。注释也可用于提供程序清单的修订情况。

对于注释，需要知道的主要一点是，它们是为使用程序清单的程序员提供的。编译器将忽略它们　在 C#中，可使用的注释有三种：

- 单行注释；
- 多行注释；
- 文档注释。

提示：注释将被编译器删除，因此在程序清单中加入注释，不会增加任何成本。有疑问时，应该包含注释

2.1.1.1　单行注释

程序清单的第 1～4 行使用的都是单行注释，第 12、18 和 22 行也是单行注释。单行注释的格式如下：

```
//comment text
```

两个斜杠表示注释的开始，直到该行的结尾都被看作注释。

单行注释不一定非得从行首开始，注释前可以有 C#代码，但从两个斜杠开始到本行的结束都属于注释。

2.1.1.2 多行注释

程序清单 2.1 没有多行注释，但有时候注释可能需要跨越多行。在这种情况下，可以在每行的开始加上两个斜杠（就像第 1～4 行那样），也可以使用多行注释。

多行注释包含开始和结束标记。要开始多行注释，可以输入斜杠和星号：

```
/*
```

该标记之后的所有内容都是注释，直到您输入结束标记为止。结束标记为一个星号和斜杠：

```
*/
```

下面为一个注释：

```
/* this is a comment */
```

下面也是一个注释：

```
/* this is
a comment that
is on
a number of
lines */
```

对于上述注释，也可以以下面的方式输入：

```
// this is
// a comment that
// is on
// a number of
// lines
```

使用多行注释的优点在于，只需加入/*和*/便可以将一段代码作为注释。编译器将/*和*/之间的所有内容都看作注释，并忽略它们。

警告： 不可以对多行注释进行嵌套，也就是说，不能在一个多行注释中包含另一个多行注释。例如，下面的注释方式是错误的：

```
/*Beginning of a comment...
/*   with another comment nested */
*/
```

2.1.1.3 文档注释

C#包含一种特殊的注释，让您能够自动创建外部文档。

这种注释使用三个斜杠，而不像单行注释那样使用两个斜杠。它还使用可扩展标记语言（XML）式的标记。XML 是一种用于标记数据的标准。虽然在 C#可以使用任何有效的 XML 标记，但常用的标记为<c>、<code>、<example>、<exception>、<list>、<para>、<param>、<paramref>、<permission>、<remarks>、<return>、<see>、<seealso>、<summary>和<value>。

文档注释被放置在代码清单中。程序清单 2.2 是一个使用文档注释的例子。您可以像以前那样

编译该清单，见第 1 天的课程“C#初步”。

程序清单 2.2　xmlapp.cs：使用 XML 注释

```
1:  // xmlapp.cs - A sample C# application using XML
2:  //          documentation
3:  //-----------------------------------------------
4:
5:  /// <summary>
6:  /// This is a summary describing the class.</summary>
7:  /// <remarks>
8:  /// This is a longer comment that can be used to describe
9:  /// the class. </remarks>
10: class myApp
11: {
12:    /// <summary>
13:    /// The entry point for the application.
14:    /// </summary>
15:    /// <param name="args"> A list of command line arguments</param>
16:     public static void Main(string[] args)
17:     {
18:        System.Console.WriteLine("An XML Documented Program");
19:     }
20: }
```

编译并执行该程序清单时，将得到如下所示的输出：

```
An XML Documented program
```

要获得 XML 文档，必须以不同于以前的方法来编译该程序清单，即在命令行进行编译时，加上参数/doc，如下所示：

```
csc /doc:xmlfile xmlapp.cs
```

这样可使以后运行程序得到的结果与前面相同。区别在于还生成了一个名为 xmlfile 的文件，其中包含 XML 格式的文档。您可以给 XML 文件取任何名称。对于程序清单 2.2，生成的 XML 文件如下：

```
<?xml version="1.0"?>
<doc>
   <assembly>
      <name>xmlapp</name>
   </assembly>
   <members>
      <member name="T:myApp">
         <summary>
         This is a summary describing the class.</summary>
         <remarks>
         This is a longer comment that can be used to describe
         the class. </remarks>
      </member>
      <member name="M:myApp.Main(System.String[])">
```

```
            <summary>
            The entry point for the application.
            </summary>
            <param name="args"> A list of command line arguments</param>
        </member>
    </members>
</doc>
```

注意：介绍XML和XML文件不在本书的范围之内。

注意：如果您使用的是诸如Visual Studion.NET工具，则需要查看文档或帮助系统，来了解如何生成XML文档。即使您使用的是这种工具，仍可以在命令行编译程序。

2.2 C#应用程序的基本组成部分

编程语言由一系列组合词组成，计算机程序以一种有机的方式使用和格式化这些单词。C#的语言的组成部分如下：

- 空白；
- C#关键字；
- 字面值；
- 标识符。

2.2.1 空白

新术语：从程序清单2.1可以知道，它被格式化了，以便代码对齐并易于阅读。程序清单中的空白区被称为*空白*（whitespace）。这一术语源自白纸上看不到空格。空白可以由空格、制表符、换行符和回车组成。

编译器几乎总是忽略空白，因此您可以添加任意数目的空格、制表符或换行符。例如程序清单2.1的第14行：

```
int radius = 4;
```

这是一个格式化良好的行，各项之间有一个空格。可以在这行代码中加入更多的空格：

```
int   radius    =   4   ;
```

其运行方式与原来相同。实际上，C#编译器编译程序时，将删除多余的空格。您还可以将上述代码格式化为跨越多行：

```
int
radius
=
4
;
```

虽然其可读性很差，但仍然能够运行。

通常情况下，由于空格将被忽略，因此可以自由地使用它来帮助格式化代码，并提高其可读性

编译器唯一不忽略的是用双引号标示的文本中的空白。对于双引号间的文本，空白至关重要，

因为其中的文本将按其原来的情况使用。在前面的程序清单中，出现过文本被双引号标示的情况。在程序清单 2.1 中，第 24 行和 25 行便包含被双引号标示的文本。这些文本将被原原本本地显示到屏幕上。

提示： 使用空白，可以提高代码的可读性。

2.2.2 C#关键字

第 1 天的课程介绍过，关键字是专用术语，它们有特殊的含义，组成了一种语言。C#语言有很多关键字，这些关键字在表 1.1 列出了。

使用 C#编程时，这些关键字有特殊的含义。阅读完本书后，您将知道这些含义。由于所有这些关键字都有特殊含义，因此被保留，您不能将其挪作他用。如果将程序清单 2.1（或本书的任何程序清单）和表 1.1 进行比较，您将发现清单的很大一部分内容是由关键字组成的。

注意： 附录C“C#关键字”包含了对每个C#关键字的简短定义。

2.2.3 字面值

新术语： *字面值是直观的硬编码值，它们的值与您看到的值一致。例如，数字4和3.14159都是字面值，另外，被标示在双引号之间的文本是字面文本。明天的课程将更详细地介绍字面值及其应用。*

2.2.4 标识符

新术语： *除了关键字和字面值外，C#程序中还可以使用其他的单词。这些单词被称为标识符。在程序清单2.1中，有很多标识符，包括第6行的System、第8行的sample、第14行的radius、第15行的PI、第16行的area以及第22行的PI、radius和area等。*

2.3 C#应用程序的结构

单词和短语用于组成句子，而句子用于组成段落。同样，空白、关键字、字面值和标识符用于组成表达式和语句，而表达式和语句组成了程序。

2.3.1 C#表达式和语句

新术语： *表达式类似于短语，是由关键字组成的代码片段。例如，下面是一个简单的范例：*

```
PI = 3.14159
PI * radius * radius
```

语句类似于句子，它们表达一个完整的意思。语句通常以标点字符分号（;）结束。在程序清单 2.1 中，第 14、15、16 行都是语句。

2.3.2 空语句

有一种语句——空语句需要特别地提一下。正如前面介绍的，语句通常以分号结束。实际上，可以让单个分号自成一行，这种语句不执行任何操作。由于其中没有任何表达式需要执行，因此这种语句被称为空语句。

您可能会问，为什么要包含一个不执行任何操作的空语句？虽然看起来您不会这样做，但阅读完本书后，您将发现空语句非常有用。第 5 天的课程将介绍空语句最流行的用途之一。

2.4 分析程序清单 2.1

学习一些概念后，有必要对程序清单 2.1 做仔细的讨论。接下来的几节将回顾一下程序清单 2.1 中的每一行代码。

2.4.1 第 1～4 行：注释

正如前面介绍的，第 1～4 行包含的是将被编译器忽略的注释。这些注释是供您和其他任何复核源代码的人使用的。

2.4.2 第 5、7、13、17、21 和 23 行：空白

第 5 行为空白行。前面介绍过，空白行只不过是空白而已，编译器将忽略它们。加入该行是为提高程序清单的可读性。第 7、13、17、21 和 23 行也是空白行，即使删除这些空白行，也不会影响程序的运行。

2.4.3 第 6 行：using 语句

第 6 行是一条语句，包含关键字 using 和字面值 System。和大多数语句一样，该语句也是以分号结束的。关键字 using 用于减少输入量。通常，using 关键字用于导入名称空间。第 6 天的课程将更详细地介绍名称空间和 using 关键字。

2.4.4 第 8 行：类声明

C#是一种面向对象编程（OOP）语言。面向对象语言使用类来声明对象。该程序定义了一个名为 sample 的类，阅读完本书后，您将理解类，而今天课程的后面将概要地介绍类。从第 6 天开始将详细介绍 C#中的类。

2.4.5 第 9、11、26 和 27 行：标点字符

第 9 行包含一个左花括号，其对应的右花括号位于第 27 行。第 11 行的左花括号与第 26 行的右花括号对应。这些括号用于包含和组织代码块。在接下来的四天中学习各种不同的命令后，您将明白如何使用这些花括号。

2.4.6 第 10 行：Main()

新术语： 计算机需要知道从哪里开始执行程序。C#程序从Main()函数（如第10行所示）开始执行。函数是一组代码，通过调用函数名称可以执行这些代码。第7天的课程将详细介绍函数。Main()函数很特别，因为它是程序的起点。

警告： 在C#应用程序中，代码的大小写至关重要。Main()中，只有M是大写。对于C和C++程序员，需要注意的是，在C#中，main()（小写的）不起作用。

2.4.7 第 14、15 和 16 行：声明

第 14、15 和 16 行包含用于创建标识符的语句，这些标识符用于存储信息。后面将使用这些标识符来完成计算。第 14 行声明了一个用于存储半径值的标识符，字面值 4 被赋给该标识符。第 15 行声明了一个用于存储圆周率的标识符 PI，其值被指定为 3.14159。第 16 行声明了一个没有被指定任何值的标识符。

2.4.8　第 20 行：赋值语句

第 20 行包含一条简单的语句，它将标识符 PI 与半径的平方相乘。然后将该表达式的结果赋给标识符 area。

2.4.9　第 24 行和 25 行：调用函数

第 24、25 行是该程序清单中最复杂的表达式。这两行调用预定义的例程，将信息输出到控制台（屏幕）。本章后面将介绍这些预定义的函数。

2.5　面向对象编程（OOP）

正如前面指出的，C#是一种面向对象的语言。为充分利用 C#，应了解面向对象语言的概念。接下来的几节将简要地介绍对象以及是什么使得一种语言成为面向对象语言的。随着阅读本书后面的内容，将知道如何在 C#中使用这些概念。

2.5.1　面向对象的概念

是什么使得一种语言成为面向对象的呢？最明显的答案是，该语言使用对象！但这一答案并不全面。第 1 天的课程介绍过，与面向对象语言相关的概念有三个：

- 封装；
- 多态；
- 继承。

还存在第四个概念：重用，这是使用面向对象语言得到的结果。

2.5.1.1　封装

封装指的是创建“包”，其中包含您所需的所有东西。对于面向对象编程而言，这意味着创建一个对象（如圆），该对象能够完成您需要对它执行的所有操作。这包括跟踪关于圆的所有信息（如半径和圆心），以及指导如何完成圆的功能，如计算半径和绘制圆。

通过封装一个圆，您让用户无需关心圆是如何工作的，而只须知道如何与圆交互。用户为何需要知道有关圆的信息在内部是如何存储的呢？只要他们能够让圆完成所需的工作，便不应该知道。

2.5.1.2　多态

多态是呈现多种形态的功能。这可用于面向对象编程的两个领域。首先，这意味着您可以以许多不同的方式调用对象和例程，而得到的结果却是相同的。以圆为例，您可能想调用圆对象来获得其面积。为此，您可以使用三点或一个点和半径。无论使用哪种方式，得到的结果都是相同的。在诸如 C 等过程性语言中，为实现这两种计算面积的方法，需要使用两个名称不同的例程。而在 C#中，仍然需要使用两个例程，但它们的名称可以相同。您只需调用该例程并传递信息，程序将自动根据传递的信息、使用正确的例程。调用例程的用户无需为到底应该使用哪个例程操心，而只要调用例程即可。

多态也能够使用多种格式，第 11 天的课程将更详细地介绍。

2.5.1.3　继承

继承是最复杂的面向对象概念。圆对象已经很好，但球对象是不是更好呢？球只不过是一种特殊的圆，它具有圆的所有特征，只不过是三维的。可以这么说，球是一个特殊的圆，具有圆的所有属性，同时还有一些其他属性。通过使用圆来创建球，就将继承圆的所有属性。能够继承属性是继承的特征之一。

2.5.1.4 重用

之所以使用面向对象语言，最重要的原因之一是重用。创建类后，您可以重用它来创建大量的对象。通过使用继承以及前面描述的其他一些特性，您可以创建出可以在大量程序中以许多方式进行重用的例程。通过封装功能，您可以创建经过测试，并被证明能够正确运行的例程。这意味着您无需测试功能是如何起作用的，而只要正确地使用它即可。这使得重用例程快速而简单。

2.5.2 对象和类

新术语：了解面向对象语言的概念后，还要了解类和对象的差别，这至关重要。类对要创建的东西做了定义，实际创建的东西是对象。简单地说，类是用于创建对象的定义。

在描述类时，经常使用的一个类比是蛋糕刀。蛋糕刀定义了蛋糕的形状，它不是蛋糕，也不能食用。它只不过是一种构造，可不断地用于制作出蛋糕。使用蛋糕刀来制作蛋糕时，您知道每个蛋糕的形状都将相同，还知道可以使用蛋糕刀制作大量的蛋糕。

和蛋糕刀一样，类也可用于创建大量的对象。例如，拥有一个圆类后，便可以使用它来创建大量的圆对象。创建一个绘制圆的程序后，便可以通过一个圆类创建出大量的圆对象。您可以将雪人中的每个圆作为一个对象，但只需要一个类来定义它们。

您还可以创建大量其他的类，包括名称、卡片、应用程序、点、圆、地址、雪人（可以使用圆类）等等。

注意： 从第6天起，将更详细地介绍类和对象。今天只是让您对面向对象的概念有个大概的了解。

2.6 显示基本信息

为使本书开始就更有趣一些，这里介绍两个可用于显示信息的例程。了解这两个例程后，您便能够显示一些基本信息。

贯穿本书，都将被用来显示信息的两个例程是：

- System.Console.WriteLine();
- System.Console.Write()。

这两个例程将信息打印到屏幕上，它们以相同的方式打印信息，只有很小的差别。WriteLine()在新的一行中打印信息，而 Writer()在打印信息时，不开始一个新行。

要显示到屏幕的信息位于括号内，如果要显示文本，则将它们用双引号标示。例如，下面的语句打印文本“Hello World”。

```
System.Console.WriteLine("Hello World");
```

注意： 第一天的课程使用过该例程。

下面的范例打印另一些文本：

```
System.Console.WriteLine("This is a line of text");
System.Console.WriteLine("This is a second line of text");
```

上述代码的输出如下：

```
This is a line of text
This is a second line of text
```

下面两行代码的输出是什么呢?

```
System.Console.WriteLine("Hello ");
System.Console.WriteLine("World! ");
```

如果您认为结果为:

```
Hello World!
```

那么您错了，结果应为:

```
Hello
World!
```

其中每个单词都位于单独的一行，如果上述两行代码中使用的是例程 Write()，则结果将如下:

```
Hello World!
```

从上面可以知道，WriteLine()在显示完文本后，将移到下一行，而 Write()不会这样做。程序清单 2.3 演示了这两个例程的用法。

程序清单 2.3　display.cs：使用 WriteLine()和 Write()

```
 1: // display.cs - printing with WriteLine and Write
 2: //---------------------------------------------
 3:
 4: class display
 5: {
 6:     public static void Main()
 7:     {
 8:         System.Console.WriteLine("First WriteLine Line");
 9:         System.Console.WriteLine("Second WriteLine Line");
10:
11:         System.Console.Write("First Write Line");
12:         System.Console.Write("Second Write Line");
13:
14:         // Passing parameters
15:         System.Console.WriteLine("\nWriteLine: Parameter = {0}", 123 );
16:
17:         System.Console.Write("Write: Parameter = {0}", 456);
18:     }
19: }
```

要在命令行编译该程序清单，请使用下面的命令:

```
csc display.cs
```

如果您使用的是集成开发环境，则可以选择相应的编译菜单项。

该程序清单的输出如下:

```
First WriteLine Line
Second WriteLine Line
First Write LineSecond Write Line
WriteLine: Parameter = 123
```

```
Write: Paramater = 456
```

分析：该程序清单的第8和9行使用System.Console.WriteLine( )来打印两段文本，从输出中可以知道，这两个文本分别位于单独的一行中 第11和12行使用的是System.Console.Write( )，这两行代码打印的文本位于同一行中，打印后没有换行。第15行和17行演示了如何在这两个例程中使用参数。

2.6.1 打印其他信息

除了打印双引号之间的文本外，还可以传递要在文本内打印的值。请看下面的范例：

```
System.Console.WriteLine("The following is a number: {0}", 456);
```

其输出如下：

```
The following is a number: 456
```

正如您看到的，“{0}”已被替换为文本后面的值，其格式如下：

```
System.Console.WriteLine("text", value);
```

其中Text可以是任何要显示的文本。“{0}”是一个值占位符，花括号表明这是占位符。“0”是一个指示器，指出使用双引号后面的第一个值。值和文本之间用逗号隔开。

要打印的文本中可以包含多个占位符，其中每个占位符使用接下来的一个序列数。要打印两个值，可以使用下面的方式：

```
System.Console.Write("Value1 is {0} and value2 is {1}", 123, "Brad");
```

上述代码的输出如下：

```
Value1 is 123 and value2 is Bard
```

本书后面将更详细的介绍这两个例程。

警告：第一个占位符的编号为0，而不是1。

注意：程序清单2.3的第15行还包含一些怪异的文本。该行中的“n\”并非错误，它是一个指示器，指出在打印后面的信息之前进行换行。第3天的课程将更详细地介绍它。

2.7 总 结

今天的课程继续为学习C#打基础。今天介绍了C#应用程序的一些基本组成部分，指出注释使得程序更易于理解。

您还知道C#应用程序的基本组成部分包括空白、C#关键字、字面值和标识符。通过一个应用程序，您知道如何将这些部分组合起来，创建一个完整的程序清单。这包括用作应用程序入口的特殊标识符：Main( )。

探讨一个程序清单后，对面向对象编程技术做了简要的介绍，包括封装、多态、继承、重用等概念。

最后对如何使用System.Console.WriteLine( )和System.Console.Write( )做了概要性介绍。这两个例程都用于将信息显示到屏幕（控制台）上。

2.8 问与答

问：基于组件和基于对象之间的区别何在？

答：有些人认为 C#是基于组件的。基于组件的开发可以看作是对面向对象编程技术的扩展。组件只不过是执行特定任务的独立代码片段。基于组件的编程需要创建大量可被重用的独立组件，您可以将它们链接起来，来创建应用程序。

问：还有哪些面向对象的语言？

答：其他面向对象的语言包括：C++、Java 和 SmallTalk。微软公司的 Visual Basic.NET 也可用于面向对象编程。还有其他的面向对象语言，只是上面列出的是最流行的。

问：何为组合（composition），这是一个面向对象术语吗？

答：许多人分不清组合和继承，但它们不是一回事。组合指的是在一个对象中使用另一个对象，图 2.1 组合了许多圆。这与前面介绍的球不同，球并不是由圆组成的，它是圆的扩展。现对组合与继承之间的区别总结如下：组合在一个类（对象）包含另一个类时发生；继承则在一个类（对象）是另一个类的扩展时发生。

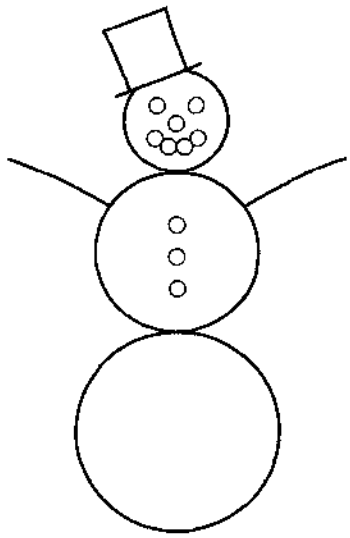


图 2.1　由多个圆组成的雪人

2.9 作　业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

2.9.1 小测验

1. 在 C#程序中，可以使用哪三种注释？
2. 如何将上述三种注释输入到 C#程序中？
3. 空白对 C#程序有何影响？
4. 下面哪些单词是关键字？

field、cast、as、object、throw、baseball、catch、football、fumble、basketball。

5. 字面值是什么?

6. 下面哪句话是正确的?

表达式是由语句组成的;

语句是由表达式组成的;

表达式和语句之间毫无关系。

7. 空语句是什么?

8. 面向对象编程最重要的概念有哪些?

9. WriteLine()和 Write()之间的区别何在?

10. 在 WriteLine()和 Write()中打印值,什么被用作占位符?

2.9.2 练习

1. 输入并编译下面的程序(listit.cs)。别忘了,请不要输入行号。

```
 1: // ListIT.cs - program to print a listing with line numbers
 2: //-------------------------------------------------------------
 3:
 4: using System;
 5: using System.IO;
 6:
 7: class ListIT
 8: {
 9:    public static void Main(string[] args)
10:    {
11:      try
12:      {
13:
14:        int ctr=0;
15:        if (args.Length <= 0 )
16:        {
17:          Console.WriteLine("Format: ListIT filename");
18:          return;
19:        }
20:        else
21:        {
22:           FileStream f = new FileStream(args[0], FileMode.Open);
23:           try
24:           {
25:             StreamReader t = new StreamReader(f);
26:             string line;
27:             while ((line = t.ReadLine()) != null)
28:             {
29:               ctr++;
30:               Console.WriteLine("{0}: {1}", ctr, line);
31:             }
32:             f.Close();
33:           }
34:           finally
```

```
35:            {
36:              f.Close();
37:            }
38:         }
39:        }
40:        catch (System.IO.FileNotFoundException)
41:        {
42:              Console.WriteLine ("ListIT could not find the file
{0}", args[0]);
43:        }
44:
45:        catch (Exception e)
46:        {
47:          Console.WriteLine("Exception: {0}\n\n", e);
48:        }
49:      }
50: }
```

2. 运行练习 1 中输入的程序，得到什么样的结果？在命令行输入下面的命令，再次运行该程序：

```
listit listit.cs
```

这次的运行结果是什么？

3. 输入、编译并运行下面的程序，该程序的运行结果是什么？

```
 1: // ex0203.cs - Exercise 3 for Day 2
 2: //---------------------------------------------
 3:
 4: class Exercise2
 5: {
 6:    public static void Main()
 7:    {
 8:       int x = 0;
 9:
10:       for( x = 1; x <= 10; x++ )
11:       {
12:           System.Console.Write("{0:D3} ", x);
13:       }
14:    }
15: }
```

4. 下面的程序有问题。输入并编译该程序，哪一行会导致错误消息？

```
 1: // bugbust.cs
 2: //---------------------------------------------
 3:
 4: class bugbust
 5: {
 6:    public static void Main()
 7:    {
```

```
 8.         System.Console.WriteLine("\nA fun number is {1}", 123 );
 9:     }
10: }
```

5. 编写将您的姓名打印到屏幕上的代码。

第 3 天课程
使用变量存储信息

开始编写程序后，您将很快发现您需要记录不同类型的信息，这可能是客户的姓名、您银行账户中的金额或您最喜欢的电影明星的年龄。为了记录这些信息，计算机程序需要使用某种方式来保存值。今天的课程将介绍以下内容：

- 变量是什么？
- 在 C#如何给变量命名？
- 使用各种数值变量。
- 字符值和数字值之间的异同。
- 如何声明和初始化变量。

3.1 变　量

变量是计算机内存中被命名的数据存储单元。在程序中使用变量名称，可以引用其中存储的信息。例如，可以创建一个用于保存数字的、名为 my_variable 的变量，在该变量中，可以存储不同的数字。

您也可以创建变量来存储非数字信息。您可以创建一个名为 BankAccount 的变量，来存储银行的账号；创建一个名为 email 的变量来存储电子邮件地址；创建一个名为 address 的变量来存储某人的邮送地址。无论所存储的信息是何种类型，变量都可以用来获得其值。

3.1.1 变量名

要在 C#程序中使用变量，您必须知道如何给变量命名。在 C#中，变量名必须遵循以下规则：

- 可以包含字母、数字和下划线（_）；
- 第一个字符必须是字母，也可以是下划线（但不推荐这样做）。下划线通常用于特殊命令，有时候其可读性不好。
- 区分大小写。C#是区分大小写的，因此名称 Count 和 count 指的是两个不同的变量。
- 不能将 C#关键字用作变量名。关键字是属于 C#语言的单词（C#关键字的完整列表，见附录 B “C#关键字”）。

下表列出了一些合法和不合法的 C#变量名范例。

变 量 名	合 法 性
Percent	合法
Y2x5_w7h3	合法
yearly_cost	合法
_2010_tax	合法，但不推荐
checking#account	非法：包含非法字符#
double	非法：是一个C#关键字
9byte	非法：第1个字符为数字

由于C#是区分大小写的，因此名称percent、PERCENT和Percent将被视为不同的变量。C#程序员通常只在变量名中使用小写字母，虽然不要求这样做。程序员常常混合使用大小写。通常，只有常量（本章后面将介绍）才全部大写。

变量名只要符合前面列出的规则即可，例如计算圆面积的程序可以将半径的值存储在一个名为radius的变量中。这种变量名清晰地说明了其用途。当然，您也可以使用名称x或billy_gates，但当其他人查看源代码时，将不知道其含义。虽然使用描述性变量名、其输入的时间可能长些，但使程序更为清晰，因此是值得的。

使用多个单词来创建变量名有许多命名约定。请看变量名circle_radius。使用下划线将变量名中的单词隔开可使其更容易理解。另一种方式叫做Pascal表示法，这种方式不使用下划线，而是将每个单词的第一个字母大写，即不是circle_radius，而是CircleRadius。另一种越来越流行的表示法是骆驼表示法（Camel notation）。骆驼表示法与Pascal表示法类似，只是变量的第一个字母是小写的而已。一种特殊的骆驼表示法叫匈牙利表示法，变量名中还包含其数据类型的信息（如是一个带小数数字还是文本），以帮助您确定存储的信息的类型。本书使用下划线，因为对大多数人来说，这种方式更容易阅读。您应该确定自己要采用的方式。

应该	不应该
应使用描述性的变量名 命令变量时，应采用并坚持一种方式	命名变量时，不要全部使用大写字母

注意：C#支持Unicode字符集，这意味着可以存储和使用任何语言中的字母，命名变量时，也可以使用任何Unicode字符。

3.2 使用变量

在使用变量之前，必须声明它。变量声明将变量的名称及其存储的信息类型告诉编译器。如果程序使用了未声明的变量，编译器将生成错误消息。

声明变量也使计算机能够为变量预留内存。通过指出变量存储的信息的具体类型，您可以获得

最佳的性能，并避免浪费内存。

3.2.1　声明变量

变量声明的格式如下：

```
typename varname;
```

其中，typename 指定了变量的类型，在接下来的几节中，您将知道 C#中可用的变量类型；varname 是变量的名称。要声明可存储标准整数的变量，可使用下面的代码行：

```
int my_number;
```

声明的变量名为 my_number，数据类型为 int。在接下来的一节中将介绍，数据类型 int 用于声明整数变量，对于本例而言，int 非常合适。

也可以在一行中声明多个同一种数据类型的变量，方法是使用逗号将变量隔开。这可以使程序清单更为简洁。请看下面的代码行：

```
int count, number, start;
```

该代码行声明了三个变量：count、number 和 start。其中每个变量的数据类型都是 int，即整型（integer）。

注意：虽然在同一行声明多个变量更为简洁，但不推荐总是这样做。有时候，使用多个声明可使代码更容易阅读和理解。分别声明对性能的影响微乎其微

3.2.2　给变量赋值

知道如何声明变量后，接下来介绍如何存储值，这很重要。毕竟变量是用来存储信息的。

将信息存储到变量中的格式如下：

```
varname = value;
```

其中，varname 是变量名，value 是被存储到变量中的值。例如，要将数字 5 存储到变量 my_variable 中，可以使用下面的代码：

```
my_variable = 5;
```

要修改变量的值，只需将新的值赋给变量即可：

```
my_variable = 1010;
```

程序清单 3.1 演示了如何给变量赋值以及修改变量的值。

程序清单 3.1　var_values.cs：给变量赋值

```
1: // var_values.cs - A listing to assign and print the value
2: //              of a variable
3: //-------------------------------------------------------
4:
5: using System;
6:
7: class var_values
8: {
9:     public static void Main()
```

```
10:     {
11:         // declare my_variable
12:         int my_variable;
13:
14:         // assign a value to my_variable
15:         my_variable = 5;
16:         Console.WriteLine("\nmy_variable contains the value {0}", my_variable);
17:
18:         // assign a new value to my_variable
19:         my_variable = 1010;
20:         Console.WriteLine("\nmy_variable contains the value {0}", my_variable);
21:     }
22: }
```

该程序清单的输出如下：

```
my_variable contains the value 5
my_variable contains the value 1010
```

分析： 在编辑器中输入上述程序清单，然后编译并执行它。有关编译和执行的方法，请参阅第1天的课程。程序清单的开始三行是注释，第11、14和18行也包含注释，注释提供信息，编译器将忽略它们。第5行将名称空间System包含进来，您完成诸如将信息写到屏幕上等工作时需要该名称空间。第7行声明了类var_values，它将是您的程序。第9行声明了程序的入口——Main()函数。记住，Main()的首字母必须大写，否则将出错。

第 12 行是今天才开始介绍的内容，它声明了一个名叫 my_variable 的整型（int）变量。该行代码被执行后，计算机将知道存在一个名为 my_variable 的变量，并让您能够使用它。第 15 行使用该变量，并将 5 赋给它。第 16 行使用 Console.WriteLine 来显示变量 my_variable 的值。从输出可以知道，显示的值为 5（在第 15 行指定的）。第 19 行将 my_variable 的值从 5 改为 1010。由于第 20 行对 Console.WriteLine 的调用打印的是新值 1010，而不是 5，因此您知道赋给变量新值的操作生效了。

在第 19 行将值赋给 my_variable 后，值 5 被覆盖了。从此以后，程序不再知道 5 曾经存在过。

注意： 使用变量之前必须声明它，但几乎可以在程序清单的任何地方进行声明。

3.2.3 初始化变量

您可能会问，是否可以在声明变量的同时给它赋值。是的，完全可以。事实上，确保在声明变量的同时初始化它，是一个很好的习惯。要在声明变量 my_variable 的同时，将其初始化为 8，可以将前面的做法组合起来：

```
int my_variable = 8;
```

使用下面的格式，可以在声明的同时初始化任何变量：

```
typename varname = value;
```

也可以在同一行声明多个变量，并给它们赋值：

```
int my_variable = 8, your_variable = 1000;
```

上述代码行声明了两个名称分别是 my_variable 和 your_variable 的整型变量，前者的值被指定为

8，后者被指定为 1000。这些声明之间以逗号分隔，而语句以分号结束。程序清单 3.2 演示了如何使用这种语句。

程序清单 3.2　multi_variable.cs：给多个变量赋值

```
 1: // multi_variables.cs
 2: // A listing to assign values to more than one variable.
 3: //-----------------------------------------------------
 4:
 5: using System;
 6:
 7: class multi_variables
 8: {
 9:    public static void Main()
10:    {
11:       // declare the variables
12:       int my_variable = 8, your_variable = 1000;
13:
14:       // print the original value of my_variable
15:       Console.WriteLine("my_variable was assigned the value {0}", my_variable);
16:
17:       // assign a value to my_variable
18:       my_variable = 5;
19:
20:       // print their values
21:       Console.WriteLine("\nmy_variable contains the value {0}", my_variable);
22:       Console.WriteLine("\nyour_variable contains the value {0}", your_variable);
23:    }
24: }
```

该程序清单的输出如下：

```
my_variable was assigned the value 8

my_variable contains the value 5

your_variable contains the value 1000
```

分析：该程序清单的第12行声明并初始化了两个变量，其中变量my_variable被初始化为8，而变量your_variable的值被初始化为1000。第15行打印变量my_variable的值，以便您知道其包含的内容。从输出可以知道，该变量包含了刚被赋给的值8。

第 18 行将值 5 赋给 my_variable。第 21 和 22 行打印两个变量最终的值。变量 my_variable 的值为 5，其原来的值永远消失了。而变量 your_variable 仍然包含的是最初的值 1000。

3.2.4　使用未初始化的变量

如果使用未初始化的变量，将会出现什么情况呢？请看下面的代码片段：

```
int blank_variable;
Console.WriteLine("\nmy_variable contains the value {0}", blank_variable);
```

在上述代码片段中，第二行打印 blank_variable。blank_variable 的值是什么呢？该变量是在第一行声明的，但没有被初始化。您将无法知道 blank_variable 的值，因为编译器将不会编译该程序。程序清单 3.3 证明了这一点。

程序清单 3.3 blank.cs：使用未被初始化的变量

```
1: // blank.cs  C Using unassigned variables.
2: // This listing causes an error!!
3: //-------------------------------------------------------
4:
5: using System;
6:
7: class blank
8: {
9:     public static void Main()
10:    {
11:        int blank_variable;
12:
13:        Console.WriteLine("\nmy_variable contains the value {0}", blank_variable);
14:    }
15: }
```

分析：该程序不会被编译，编译器将给出如下所示的错误消息：

```
blank.cs(13,67): error CS0165: Use of unassigned local variable 'blank_variable'
```

C#不允许使用未被初始化的变量。

注意：在诸如C和C++等其他语言中，上述程序清单将被编译，在这种情况下，打印的blank_variable的值将是无用信息。C#能够防止这种错误发生。

3.3 了解计算机内存

对于知道计算机内存如何工作的读者，可以跳过本节；如果没把握，则请阅读之。本节的内容有助于您理解编程技术。

计算机中的 RAM 有何用途呢？其用途有多种，但对程序员而言，需要关心的只有数据存储。数据是 C#程序使用的信息。您的程序是否在维护一个联络清单，监控股票市场，规划预算，跟踪商品的价格。正在运行的程序所使用的信息（姓名、股价、开支、价格）被保存在计算机内存的变量中。

计算机运行时，使用随机存储器（RAM）来存储信息。RAM 位于计算机内的集成电路或芯片中。RAM 是易失性的，这意味着需要时，其中的信息将被擦除，并替换为新的信息；而且，只有当计算机开启时，RAM 才具备记忆功能，当计算机关闭后，其中的信息将丢失。

字节是计算机数据存储的基本单位。所有的计算机都安装了一定数量的 RAM。系统中 RAM 的量通常以 MB（兆字节）为单位，如 32MB、64MB、128MB 等。1MB 等于 1024KB（千字节），而 1KB 为 1024 字节。因此，一个具有 8MB 内存的系统实际上有 8192（8 × 1024）KB 的 RAM，即 8388608（8192 × 1024）字节。表 3.1 列出了存储某些类型的数据所需要的内存量（单位为字节）。

表 3.1　存储数据所需的内存空间

数　据	需要的内存空间（字节）
字母 x	2
数字 500	2
数字 241.105	4
字符串 Teach Yourself C#	34
打字机中一页的内容	约 4000

计算机中的 RAM 是顺序排列的，一个字节跟着一个字节。每个字节都有用于标识它的唯一性地址——一个用来将它与内存中其他字节区分开来的地址。内存单元被依次指定地址，从 0 开始，直到系统的最大内存。就现在而言，您无需关心地址，地址是自动处理的。

粗略地了解一些关于内存的知识后，我们回到 C#编程技术，介绍 C#如何使用内存来存储信息。

3.4　C#数据类型

您已经知道如何声明、初始化变量以及如何修改变量的值。知道可以使用哪些数据类型至关重要。前面介绍过，声明变量时必须指定其数据类型。您已经知道，使用关键字 int 声明的变量可以存储整数。整数是不包含小数部分的数字。到目前为止，您声明的只是用来存储整数的变量，如果要存储诸如小数和字符等其他类型的数据，该如何办呢？

3.5　数值变量类型

C#提供了多种不同的数值变量类型。您需要不同类型的变量，因为不同的数值所需的内存空间不同，对其执行数学运算的难易程度也不同。小型整数（如 1199 和-8）需要的内存空间较少，计算机对其执行数学运算（如加、乘等）时速度较快；而大型整数以及包含小数部分的数值需要的内存空间较多，对其执行数学运算所需的时间也更长。通过使用合适的变量类型，可以确保程序的运行效率尽可能高。

接下来的几节将数值数据类型分为 4 类：

- 整数；
- 浮点数；
- Decimal；
- 布尔型。

本章前面介绍过，变量被存储在内存中；而且，不同类型的信息需要的内存量也不同。存储变量所需的内存空间取决于数据类型。程序清单 3.4 中的一些代码以前没有介绍过，但它让您知道存储一些 C#数据类型所需要的内存空间。

新术语：编译该程序清单时，必须包含一些提供给编译器的额外信息。这种信息被称为编译器*标记*（flag），可以在命令行中包含它们。具体地说，需要添加/unsafe 标记，如下所示：

```
csc /unsafe sizes.cs
```

如果使用的是集成开发环境，则需要按其文档说明设置 unsafe 选项。

程序清单 3.4 size.cs：各种数据类型所需的内存空间

```
 1: // sizes.cs--Program to tell the size of the C# variable types
 2: //---------------------------------------------------------
 3:
 4: using System;
 5:
 6: class sizes
 7: {
 8:    unsafe public static void Main()
 9:    {
10:       Console.WriteLine( "\nA byte    is {0} byte(s)", sizeof( byte ));
11:       Console.WriteLine( "An sbyte   is {0} byte(s)", sizeof( sbyte ));
12:       Console.WriteLine( "A char    is {0} byte(s)", sizeof( char ));
13:       Console.WriteLine( "\nA short   is {0} byte(s)", sizeof( short ));
14:       Console.WriteLine( "An ushort  is {0} byte(s)", sizeof( ushort ));
15:       Console.WriteLine( "\nAn int    is {0} byte(s)", sizeof( int ));
16:       Console.WriteLine( "An uint   is {0} byte(s)", sizeof( uint ));
17:       Console.WriteLine( "\nA long    is {0} byte(s)", sizeof( long ));
18:       Console.WriteLine( "An ulong  is {0} byte(s)", sizeof( ulong ));
19:       Console.WriteLine( "\nA float   is {0} byte(s)", sizeof( float ));
20:       Console.WriteLine( "A double  is {0} byte(s)", sizeof( double ));
21:       Console.WriteLine( "\nA decimal is{0}byte(s)", sizeof( decimal));
22:       Console.WriteLine( "\nA boolean  is {0} byte(s)", sizeof( bool ));
23:    }
24: }
```

该程序清单的输出如下：

```
A byte     is 1 byte(s)
An sbyte    is 1 byte(s)
A char     is 2 byte(s)

A short    is 2 byte(s)
An ushort   is 2 byte(s)

An int     is 4 byte(s)
An uint    is 4 byte(s)

A long    is 8 byte(s)
An ulong   is 8 byte(s)

A float    is 4 byte(s)
A double   is 8 byte(s)

A decimal  is 16 byte(s)

A boolean  is 1 byte(s)
```

分析：虽然还未介绍这些数据类型，但我认为在这里提供该程序清单很有用。在阅读接下来的几节时，可以参考该程序清单及其输出。

该程序清单使用了 C#关键字 sizeof，该关键字指出变量的大小。在该程序清单中，sizeof 被用来显示不同数据类型的大小。例如要确定 int 的大小，可以使用下面的代码：

```
sizeof(int)
```

如果您声明了一个名为 x 的变量，则可以使用下面的代码来确定其大小——实际上是其数据类型的大小：

```
sizeof(x)
```

程序清单 3.4 的输出列出了存储每种 C#数据类型所需的字节数。对于 int，需要 4 个字节的存储空间，而 short 需要两个字节。使用的内存量决定了可以存储的数字的大小。接下来的几节将更详细地进行介绍。

关键字 sizeof 并不太常用，但对于说明本章的内容很有用。关键字 sizeof 进入到内存中，以确定变量或数据类型的大小。在 C#中，应避免直接进入到内存中。第 8 行额外加入了关键字 unsafe。如果不加入该关键字，则编译该程序时将出错。就现在而言，只需知道加入关键字 unsafe 的原因在于关键字 sizeof 直接进入内存。

警告：可以使用C#关键字sizeof，但通常应避免使用它。关键字sizeof有时候会直接访问内存，以确定数据类型的大小。在纯粹的C#程序中，应避免直接访问内存。

3.5.1　整型数据类型

到目前为止，您只使用过一种整型数据类型：int。整型数据类型用于存储整数。整数指的是任何不包含小数部分的数值。数值 1、1000、56 万亿和–534 都是整型值。

C#提供了 9 种整型数据类型，如下所示：

- 整数（int 和 uint）；
- 短整数（short 和 ushort）；
- 长整数（long 和 ulong）；
- Byte（byte 和 sbyte）；
- 字符（char）。

3.5.1.1　整数

从程序清单 3.4 知道，整数占用 4 个字节的内存。这包括数据类型 int 和 uint。本书前面的很多程序都使用过 int 数据类型。虽然您可能还不了解这种数据类型，但它并不能存储任何数字，它只能存储使用 4 个字节（32 位）能够表示的符号整数，即–2147483648 到 2147483647 之间的任何整数。

Int 类型的变量是带符号的，即可以为正，也可以为负。从技术上说，4 个字节可以存储大到 4294967295 的整数，但使用 32 位中的一位表示正负后，可以存储的值便只能大到 2147483647，而小到–2147483648。

前面介绍过，信息的存储单位叫字节。一个字节实际上是由 8 位组成的，位是计算机最基本的存储单位。位的取值有两个：0 或 1。使用位和二进制，可以将数字存储在多个位中。附录 C 将详细介绍二进制。

要使 int 能够存储更大的值，可以让它不带符号。无符号整数只能是正的。这样做的好处显而

易见。uint 数据类型用于声明无符号整数，其存储的值为 0～4294967295。

如果您试图存储一个更大的数字，将出现什么情况呢？如果在 int 或 uint 变量中存储带小数的值，情况将如何呢？如果在 uint 变量中存储负数，情况又将如何呢？程序清单 3.5 对上述问题做出了回答。

程序清单 3.5 int_conv.cs：执行非法操作

```
 1: // int_conv.cs
 2: // storing bad values. Program generates errors and won't compile.
 3: //-----------------------------------------------------------------
 4:
 5: using System;
 6:
 7: class int_conv
 8: {
 9:    public static void Main()
10:    {
11:       int val1, val2;   // declare two integers
12:       uint pos_val;     // declare an unsigned int
13:
14:       val1 = 1.5;
15:       val2 = 9876543210;
16:       pos_val = -123;
17:
18:       Console.WriteLine( "val1 is {0}", val1);
19:       Console.WriteLine( "val2 is {0}", val2);
20:       Console.WriteLine( "pos_val is {0}", pos_val);
21:    }
22: }
```

该程序清单的编译输出如下：

```
int_conv.cs(14,15): error CS0029: Cannot implicitly convert type 'double' to 'int'
int_conv.cs(15,15): error CS0029: Cannot implicitly convert type 'long' to 'int'
int_conv.cs(16,18): error CS0031: Constant value '-123' cannot be converted to a 'uint'
```

警告：该程序将出现编译错误。

分析：该程序不能通过编译。正如您看到的，编译器捕获了所有的错误。第14行试图将包含小数的值赋给一个整数变量；而第15行试图将一个过大的值赋给整数变量。记住，int变量能够存储的最大值为2147483647。最后，第16行试图将一个负数赋给一个无符号整数（uint）变量。正如输出所表明的，编译器捕获了所有这些错误，并阻止程序被编译。

3.5.1.2 短整型

int 和 uint 变量使用 4 个字节的内存，通常不需要存储这么大的整数，例如在记录星期几（1～7）、年龄、烘烤蛋糕的温度时，并不需要这么大的数。

当您需要存储整数，而又想节省内存时，可以使用 short 和 ushort 数据类型。和 int 一样，short 也用于存储整数；不同的是，它只占用 2 个字节，而不是 4 个字节。从程序清单 3.4 可以知道，对于 short 和 ushort，sizeof 的返回值都是 2。如果要存储的整数有正有负，则可以使用 short；如果只有

正数，同时想利用额外的空间，则可以使用 ushort。Short 能够存储的整数范围为–32768 ~ 32767；而 ushort 为 0 ~ 65535。

3.5.1.3　长整型

如果值太大，无法使用 int 和 uint 存储，则可以使用另一种数据类型 long。与 short 和 int 一样，long 数据类型也有一种无符号形式，叫做 ulong。从程序清单 3.4 的输出可以知道，long 和 ulong 占用的内存都为 8 个字节，这使它们能够存储非常大的整数，其中 long 可以存储的整数范围为 –9223372036854775808 ~ 9223372036854775807，而 ulong 为 0 ~ 18446744073709551615。

3.5.1.4　Byte

您可以将整数存储为占用 2、4 或 8 个字节内存的数据类型。当您需要的整数非常小时，可以使用单个字节来存储。为简化问题，使用 1 个字节内存的数据类型被称为 byte。和前面的整型数据类型一样，byte 也有带符号形式（sbyte）和无符号形式（byte）。Sbyte 的取值范围为–128 ~ 127；而 byte 为 0 ~ 255。

3.5.1.5　字符

除了数字外，您还经常需要存储字符。字符指的是字母，如 A、B、C，乃至扩展字符，如笑脸。您可能需要存储的其他字符，如$、%、*等。您甚至还需要存储外语中的字符。

计算机并不能识别字符，而只能识别数字。为此，所有的字符都被存储为数值。为确保所有的人都使用相同的值，制定了一种名叫 Unicode 的标准。在 Unicode 中，每个字符和符号都使用一个整数表示，这也是字符数据类型被视为整型的原因所在。

为指明数字将被看作字符，可以使用数据类型 char。char 是一个占用两个字节内存的数字，被解释为一个字符。程序清单 3.6 是一个使用 char 的程序。

程序清单 3.6　chars.cs：使用字符

```
// chars.cs
// A listing to print out a number of characters and their numbers
//-----------------------------------------------------------------

using System;

class chars
{
   public static void Main()
   {
      int ctr;
      char ch;

      Console.WriteLine("\nNumber   Value\n");

      for( ctr = 60; ctr <= 95; ctr = ctr + 1)
      {
         ch = (char) ctr;
         Console.WriteLine( "{0} is {1}", ctr, ch);
      }
   }
}
```

该程序清单的输出如下所示：

```
Number Value

60 is <
61 is =
62 is >
63 is ?
64 is @
65 is A
66 is B
67 is C
68 is D
69 is E
70 is F
71 is G
72 is H
73 is I
74 is J
74 is K
76 is L
77 is M
78 is N
79 is O
80 is P
81 is Q
82 is R
83 is S
84 is T
85 is U
86 is V
87 is W
88 is X
89 is Y
90 is Z
91 is [
92 is \
93 is ]
94 is ^
95 is _
```

分析：该程序清单显示一定范围内的数值及其对应的字符。第11行声明了一个名叫ctr的整数变量，用于遍历一系列的整数。第12行声明了一个名为ch的字符变量。第14行为要显示的信息打印标题。

第 16 行包含了一些新内容。就现在而言，无需完全理解该行代码。第 5 天的课程将详细介绍这种语句。现在只需要该行代码将 ctr 的值设置为 60，然后运行第 18～19 行，再将 ctr 的值加 1。这一过程将一直重复下去，直到 ctr 的值大于 95 为止。最终结果是，第 18～19 行将重复运行，而 ctr 的值将分别为 60、61、62...，直到 ctr 的值为 95 为止。

第 18 行将 ctr 的值（开始为 60）赋给字符变量 ch。由于 ctr 是一个整数，因此您需要命令计算

机将其转换为字符，这是通过(char)语句实现的。关于这一点，后面将做更详细的介绍。

第 19 行打印存储在 ctr 和 ch 中的值，正如您看到的，整数变量 ctr 是作为数字打印的，而 ch 的值是作为字符（而不是数字）打印的　从该程序清单的输出可以知道，字符 A 被表示 为值 65，而字符 B 对应的值为 66。

注意：计算机实际上只能识别（位于比特中的）1和0，它分别将它们看作是开和关（或电压为正和负）。二进制使用0和1来表示数字，附录C对二进制进行了介绍。

字符字面值

如何将字符赋给 char 变量呢？将字符放在单引号中。例如，要将字符 a 赋给变量 my_char，可以使用下面的语句：

```
my_char = 'a';
```

除了常规字符外，您还可能需要使用一些扩展字符。在很多程序清单中，已经使用过一个扩展字符：\n，该字符用于换行。表 3.2 列出了一些最常用的扩展字符，程序清单 3.7 演示了一些特殊字符的用法。

表 3.2　　扩展字符

字　符	含　义
\b	空格
\n	换行符
\t	水平制表符
\\	反斜杠
\'	单引号
\"	双引号

注意：表3.3列出的扩展字符常被称为转移字符，因为反斜杠避开了常规文本，指出后面的字符为特殊（扩展）字符。

程序清单 3.7　chars_table.cs：特殊字符

```
 1: // chars_table.cs
 2: //-------------------------------------------------------------
 3:
 4: using System;
 5:
 6: class chars_table
 7: {
 8:    public static void Main()
 9:    {
10:       char ch1 = 'Z';
11:       char ch2 = 'x';
12:
13:       Console.WriteLine("This is the first line of text");
```

```
14:       Console.WriteLine("\n\n\nSkipped three lines");
15:       Console.WriteLine("one\ttwo\tthree <-tabbed");
16:       Console.WriteLine(" A quote: \' \ndouble quote: \"");
17:       Console.WriteLine("\n ch1 = {0}   ch2 = {1}", ch1, ch2);
18:    }
19: }
```

该程序清单的输出如下：

```
This is the first line of text



Skipped three lines
one    two    three <-tabbed
 A quote: '
Double quote: "

 ch1 = Z   ch2 = x
```

分析：该程序清单说明了两个概念。首先，第10和11行演示了如何将字符赋给char变量——只需将字符放在单引号中即可。第13～17行演示了如何使用扩展字符。第13行没有什么特别的地方，第14行打印三个换行符，然后打印一些文本。第15行打印one、two和three，并在它们之间加上制表符。第16行打印一个单引号和一个双引号，该行的结尾处连续出现了两个双引号。最后，第17行打印了ch1和ch2的值。

3.5.2 浮点数

并非所有的数字都是整数。使用带小数的数字时，需要使用其他的数据类型。和存储整数一样，根据要使用的数字的大小以及内存空间的多少，您可以使用不同的数据类型来存储浮点数。其中最主要的两种是 float 和 double。

3.5.2.1 float

float 用于存储带小数的数值。例如，计算圆的周长或面积时，得到的结果常常不是整数。存储诸如 1.23 或 3.14159 这样的数字时，需要使用非整型数据类型。

float 数据类型使用 4 个字节的内存来存储数字，因此其可以存储的数值范围大约为 1.5×10^{-45} 到 3.4×10^{38}。

注意：10^{38}相当于执行37次10×10，得到的结果为1后面跟38个零；而10^{-45}相当于执行46次10÷10，得到的结果为在小数点和1之间有44个0。

警告：float的精度为7位，因此float的值经常会有些误差。例如，将10.00减去9.90得到的结果可能不是0.10，而是一个接近于0.099999999的值。这种舍入误差通常是微不足道的。

3.5.2.2 double

double 类型的变量占用 8 个字节的内存，这意味着其取值可以比 float 大得多。Double 的取值大约为 5.0×10^{-324}到 1.7×10^{308}，其精度通常为 15～16 位。

注意：C#支持IEEE 754格式的4字节精度（32位）和8字节精度（64位），因此某些数学函数能够返回非常精确的值。如果您执行除以0的运算，结果将为无穷大（可以是正无穷大或负无穷大），

如果将0除以0，得到的结果将为Not-a-Number。0可以为正，也可以为负　更详细的信息请参阅C#文档。

3.5.3 Decimal

C#提供了另一种可用于存储特殊小数的数据类型：decimal。这种数据类型是为存储精度更高的数字而设计的。将数字存储在 float 和 double 变量中时，将引入舍入误差。例如，将 10.00 减去 9.90 的结果存储在 double 变量中时，得到的值将为 0.09999999999999645.而不是 0.10。如果使用 decimal 变量来执行这种数学运算，则得到的结果将为 0.10。

提示：如果您计算的是金额，或执行的是精度非常重要的金融方面的计算时，则应该使用 decimal，而不是float或double。

decimal 变量占用 16 个字节的内存。与其他数据类型不同的是，decimal 没有无符号形式。decimal 变量可存储的数值范围为 1.0×10^{-28} 到 7.9×10^{28}，其精度为 28 位。

3.5.4 布尔型

最后一种简单的数据类型是布尔型。有时候，您需要知道某种东西是开还是关、真还是假、是还是否。布尔型数字通常被设置为两个值之一：0 或 1。

C#中的布尔型数据类型为 bool。从程序清单 3.4 的输出中可以知道，bool 使用 1 个字节的内存。bool 变量的取值只能是 true 或 false，true 或 false 是 C#关键字。这意味着 bool 数据类型只能存储 true 或 false。

警告：“Yes”、“no”、“on”、“off”都不是C#关键字，这意味着您不能将布尔型变量设置为这些值，而必须设置为true或false。

3.5.5 检查和不检查

通过本章前面的内容，您知道如果将一个过大的值赋给变量，将出错。有时候，您可能不希望出现这种错误。在这种情况下，您可以让编译器不对代码进行检查。这是通过关键字 unchecked 实现的，程序清单 3.8 对此进行了演示。

程序清单 3.8　unchecked.cs：命令编译器不对代码进行检查

```
1: // unchecked.cs
2: //-----------------------------------------------------------------
3:
4: using System;
5:
6: class sizes
7: {
8:    public static void Main()
9:    {
10:       int val1 = 2147483647;
11:       int val2;
12:
13:       unchecked
14:       {
15:          val2 = val1 + 1;
16:       }
17:
```

```
18:        Console.WriteLine( "val1 is {0}", val1);
19:        Console.WriteLine( "val2 is {0}", val2);
20:    }
21: }
```

该程序清单的输出如下：

```
var1 is 2147483647
var2 is -2147483648
```

分析：该程序清单的第13行使用了unchecked，编译器将不检查第14行和第16行的花括号之间的代码，编译该程序清单时，不会出现错误，运行该程序清单时，将得到一个怪异的结果。2147483647是带符号int变量能够存储的最大值，第10行将这个值赋给了var1。第15行（不被检查的行）将var1能够存储的最大值与1相加，并将结果赋给var2。由于该行不会被检查，因此程序将继续运行。结果是，存储在var2中的值为最小的负数。

这类似于汽车中的里程表，当里程达到最大（如 999999）时，再运行 1 公里后，里程表的值便变成 000000 了。此时，汽车并非一辆新车，而是一辆里程超过里程表最大值的汽车。在这种情况下，变量不会变为 0，而是它能够存储的最小值。在上述程序清单中，值变为-2147483648。

请将程序清单的第 13 行修改为如下所示，然后编译并运行它。

```
13:     checked
```

程序将被编译，但能否运行呢？执行该程序时将出错。如果系统询问是否运行调试器，请不要运行。出现的错误与下面类似：

```
Exception occurred: System.OverflowException: An exception of type System.OverflowException
was thrown.
at sizes.Main( )
```

本书后面将介绍如何处理程序中的这种错误。现在您只需记住，如果您认为有可能赋给变量一个非法值时，应启用检查功能。

3.5.6 简单数据类型

前面介绍的 C#数据类型被称为简单数据类型。简单数据类型包括 sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、bool 和 decimal。第 1 天和第 2 天的课程介绍过，C#程序运行在 CLR 上。这些数据类型中的每一种直接对应于 CLR 使用一种数据类型，它们之所以被称为简单数据类型是因为，它们与 CLR 中的类型有直接关系，从而与.NET 框架中的类型有直接关系。表 3.3 列出了 C#数据类型对应的.NET 数据类型。

表 3.3　　C#和.NET 数据类型

C#数据类型	.NET 数据类型
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.Uint16

续表

C#数据类型	.NET 数据类型
int	System.Int32
uint	System.Uint32
long	System.Int64
ulong	System.Uint64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

如果要使用对应的.NET 数据类型声明一个变量（虽然没有理由这样做），则可以使用下面的代码：

```
System.Int32 my_variable = 5;
```

从中可以知道，System.Int32 比使用 int 要复杂得多。程序清单 3.9 演示了如何使用.NET 数据类型。

程序清单 3.9　net_vars.cs：使用.NET 数据类型

```
// net_vars
// Using a .NET data declaration
//-----------------------------------------------

using System;

class net_vars
{
    public static void Main()
    {

        System.Int32 my_variable = 4;
        System.Double PI = 3.1459;

        Console.WriteLine("\nmy_variable is {0}", my_variable );
        Console.WriteLine("\nPI is {0}", PI );
    }
}
```

该程序清单的输出如下：

```
my_variable is 4
```

```
PI is 3.14159
```

第 12 行和 13 行分别声明了一个 int 和 double 变量。第 15 和 16 行打印了这些变量的值。.NET 类型的所有功能都可以使用更简单的 C#命令实现。但需要明白的是，简单的 C#数据类型将被转换为对应的.NET 类型。您将发现，使用.NET 类型的其他所有编程语言也包含将被转换为.NET 类型的数据类型。

注意： 通用类型系统（Common Type System，或称CTS）是一套CLR 中的数据类型都必须遵守 的规则 C#中的简单数据类型和.NET数据类型都遵守这些规则。如果某种语言在创建数据类型时遵守了CTS，则它创建和存储的数据将能够与其他也遵循了CTS的编程语言兼容。

3.6 字面值和变量

新术语： 在至今为止介绍的范例中，使用了大量并非变量的数字和值。您经常需要在源代码中输入数字或值。在源代码中，*字面值*代表的是其本身。例如，在下面的代码行中，数字 10 和字符串 "Bob is a fish" 都是字面值。正如您看到的，字面值可以赋给变量。

```
int x = 10;
myStringValue = "Bob is a fish";
```

3.6.1 数值型字面值

前面的很多范例使用过数值型字面值。默认情况下，数值型字面值的数据类型为 int 或 double；如果是整数，则为 int 型，如果是浮点数，则为 double 型。例如下面的代码：

```
nbr = 100;
```

这里，100 是一个数值型字面值。默认情况下，其数据类型被视为 int，而不管变量 nbr 是什么数据类型。现在来看下面的例子：

```
nbr =99.9;
```

在这个例子中，99.9 也是一个数值型字面值，但其数据类型默认为 double，而不管 nbr 是什么数据类型，虽然 99.9 可作为 float 类型存储。在下面的代码行中，100.的数据类型是 int 还是 double 呢？

```
x = 100.;
```

这是一个比较困难的问题。如果您的猜测是 int，则是错误的。由于 100.中包含小数点，因此它是一个 double。

3.6.1.1 整型字面值的默认类型

当您使用整型值时，系统将根据其大小将其视为 int、uint、long 或 ulong。如果该值可以以 uint 或 int 保存，则被视为相应的数据类型。否则，将被视为 long 或 ulong 类型。要指定字面值的数据类型，可以使用字面值后缀。例如，要将数字 10 指定为一个 long 型的字面值（有符号或无符号），可以这样编写代码：

```
10L;
```

可以使用 u 或 U 来指定无符号型的值。如果要指定一个无符号长整型值，可以结合使用两个后缀：ul。

注意： 如果您使用小写的l来声明long字面值，微软公司的C#编译器将提出警告。该编译器提出警告是提醒您，小写l容易与数字1混淆。

3.6.1.2　浮点数字面值的默认类型

正如前面指出的，默认情况下，浮点数字面值被视为 double 类型。要将字面值指定为 float 类型，可以在数字后面加上 f 或 F。例如，要将数字 4.4 赋给 float 变量 my_float，可以使用下面的方法：

```
my_float = 4.4f;
```

要将字面值指定为 decimal 类型，可以使用后缀 m 或 M。例如，下面的代码行声明 my_decimal 的值等于 decimal 型数值 1.32。

```
my_decimal = 1.32m;
```

3.6.2　布尔型字面值

布尔型字面值已经介绍过了，true 和 false 都是字面值，同时它们也是 C#关键字。

3.6.3　字符串字面值

将字符组合在一起，可以形成单词、短语和句子。在编程用语中，一组字符被称为字符串。字符串是可以识别出来的，因为它们位于一对双引号之间。以前的例子中也使用过字符串，例如 Console.WriteLine 例程便是用了一个字符串。字符串字面值指的是一对双引号之间的一组字符。下面都是字符串：

```
"Hello, World!"
"My Name is Bradley"
"1234567890"
```

最后一个例子之所以是字符串字面值，而不是数值型字面值，是因为这些数字位于引号之间。

注意： 在字符串中可以使用表3.3中的任何特殊字符

3.7　常　量

除了使用字面值外，有时候您可能想将一个值存储到变量中，并固定它。例如，如果您声明了一个名为 PI 的变量，将其值设置为 3.14159，则您希望其值始终为 3.14159。没有理由再去修改它的值。另外，您还想防止其他人修改它。

要声明一个存储常量的变量，可以使用关键字 const。例如，要将 PI 声明为一个固定值，可以使用下面的代码：

```
const PI = 3.14159;
```

您可以在程序中使用 PI，但不能修改它的值。关键字 const 固定了其内容。可以针对任何数据类型的任何变量使用关键字 const。

提示： 为使常量易于识别，可以将其名称中的字母全部采用大写。这样，将很容易知道某个变量的值是固定的。

3.8 引用类型

至此，已经介绍了很多数据类型。C#提供了两种主要的信息存储方式：按值（byval）和按引用（byref）。前面介绍的基本数据类型都是按值来存储信息的。

当变量按值存储信息时，变量将包含实际的信息。例如，当您将 123 存储到变量 x 中后，x 的值便是 123。变量 x 实际上包含了值 123。

按引用存储信息要稍微复杂些。如果变量按引用存储，而不是存储信息本身，则变量存储的将是信息所在的位置。换句话说，它存储的是到信息的引用。例如，如果 x 是一个“按引用”变量，则它包含的将是关于值 123 所在位置的信息，它并不存储值 123 本身。图 3.1 说明了这种差别。

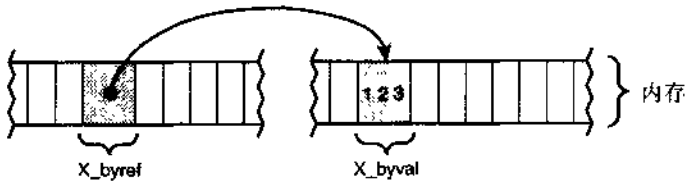


图 3.1 按引用和按值

在 C#使用的数据类型中，按引用存储的有：

- 类；
- 字符串；
- 接口；
- 数组；
- 代表（Delegate）。

本书后面将详细介绍上述所有数据类型。

3.9 总 结

今天的课程介绍了计算机如何存储信息。重点介绍了按值存储数据的数据类型，包括 int、uint、long、ulong、bool、char、short、ushort、float、double、decimal、byte 和 sbyte。除此之外，还介绍了如何声明和命名变量以及如何设置变量的值，包括使用字面值。表 3.4 列出了这些数据类型及其相关的信息。

表 3.4 C#数据类型

C#数据类型	对应的.NET 数据类型	占用的字节数	最小值	最大值
sbyte	System.SByte	1	-128	127
byte	System.Byte	1	0	255
short	System.Int16	2	-32768	32767
ushort	System.Uint16	2	0	65535
int	System.Int32	4	-2147483648	2147483647

续表

C#数据类型	对应的.NET 数据类型	占用的字节数	最小值	最大值
uint	System.Uint32	4	0	4294967295
long	System.Int64	8	–9223372036854775808	9223372036854775807
ulong	System.Uint64	8	0	18446744073709551615
char	System.Char	2	0	65535
float	System.Single	4	1.5×10^{-45}	3.4×10^{38}
double	System.Double	8	5.0×10^{-324}	1.7×10^{308}
bool	System.Boolean	1	false(0)	true(1)
decimal	System.Decimal	16	1.0×10^{-28}	约 7.9×10^{28}

3.10　问与答

问：为何不将所有的数值都声明为较大的数据类型，而是较小的数据类型？

答：虽然使用较大的数据类型好像是合理的，但这样做效率不高。您使用的系统资源（内存）不应该超过需求。

问：将一个负数赋给一个无符号变量将出现什么情况？

答：如果使用的是字面值将出错，编译器指出您不能将负数赋给无符号变量；如果是由于计算而导致无符号变量的值小于 0，得到的结果将是错误的。本书后面将介绍如何检查这种错误值。

问：decimal 值的精度高于 float 和 double 值，在不同的数据类型之间进行转换时，将如何进行舍入？

答：将 float、double 或 decimal 转换为某种整型变量类型时，值将被舍入。如果数字过大，无法存储到变量中，将出错。

将 double 数据转换为 float 型时，如果数据过大或过小，则将分别表示为无穷大和 0。

将 float 或 double 数值转换为 decimal 型时，数值将被舍入。只有 28 位小数之后的值才会被舍入，并且只有在必要时才会舍入。如果被转换的值太小，无法表示为 decimal，则转换后的值将为 0；如果值太大，无法以 decimal 存储，则将发生错误。

将 decimal 值转换为的 float 或 double 时，将被舍入为与之最接近的 float 或 double 值。记住，decimal 的精度高于 float 或 double，转换后，精度将降低。

问：还有哪些语言也遵循 CLR 中的 CTS？

答：Microsoft Visual Basic.NET（第 7 版）和 Microsoft Visual C++.NET（第 7 版）都支持 CTS。另外，许多语言的版本也支持 CTS，包括 Python、COBOL、Perl、Java 等等。有关支持 CTS 的其他语言，请查阅微软公司的网站。

3.11 作 业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

3.11.1 小测验

1. C#中的按值数据类型有哪些?
2. 带符号变量和无符号变量之间的区别何在?
3. 要存储数值 55，可使用的最小数据类型是什么?
4. short 变量能够存储的最大值是多少?
5. 字符 B 对应的数值是什么?
6. 一个字节包含多少位?
7. 哪些字面值可以被赋给布尔型变量?
8. 指出三种引用数据类型。
9. 哪种浮点数据类型的精度最高?
10. 在.NET 中，与 C#中的 int 数据类型对应的是什么数据类型?

3.11.2 练习

1. 修改程序清单 3.6 的取值范围，以打印小写字母。
2. 编写一行代码，声明一个名为 xyz 的 float 变量，并将值 123.456 赋给该变量。
3. 下述哪些变量名是合法的?

a) x

b) PI

c) 12months

d) sizeof

e) nine

4. 下面的程序有问题，请在编辑器中输入该程序，并编译它。哪一行会导致错误消息?

```
 1: //Bug Buster
 2: //-------------------------------------------------------
 3: using System;
 4:
 5: class variables
 6: {
 7:     public static void Main()
 8:     {
 9:         double my_double;
10:         decimal my_decimal;
11:
12:         my_double = 3.14;
13:         my_decimal = 3.14;
14:
15:         Console.WriteLine("\nMy Double:  {0}", my_double);
16:         Console.WriteLine("\nMy Decimal: {0}", my_decimal);
17:
```

```
18:     }
19: }
```

5. 选做题：编写一个程序，其中对于每种数据类型，都声明两个变量、并分别将值 10 和 1.879 赋给它们。

第 4 天课程
使用运算符

知道如何在变量中存储信息后，您可能想使用其中的数据。您很可能需要通过修改来操纵数据。例如，您可能需要使用圆的半径来计算出圆的面积。今天将学习以下内容：

- C#中运算符的类型和分类；
- 使用不同的运算符来操纵信息；
- 使用 if 命令修改程序的流程；
- 了解运算符的优先级；
- 探索按位运算——如果您足够勇敢的话。

4.1 运算符的类型

运算符用于操纵信息。在前几天的范例中，您已经使用过运算符。运算符用于执行诸如加法、乘法、比较等运算。

运算符可以分成很多类：

- 基本的赋值运算符；
- 数学/算术运算符；
- 关系运算符；
- 条件运算符；
- 其他运算符（类型、大小）。

今天将详细介绍各类运算符以及其中的每个运算符。除了运算符类别外，理解运算符语句的结构也很重要。运算符结构有三种：

- 单目；
- 双目；
- 三目。

4.1.1 单目运算符

单目运算符指的是只影响一个变量的运算符。例如，要将 1 变成负值，您输入如下代码：

```
-1
```

如果有一个名为 x 的变量，要将其变成负的，可以使用下面的代码：

```
-x
```

负运算符只需要一个变量，因此是单目的。单目运算符的使用格式为下面两种之一，具体使用哪种格式取决于运算符：

```
[operator][variable]
```

或

```
[variable][operator]
```

4.1.2 双目运算符

单目运算符只使用一个变量，而双目运算符使用两个变量。例如，加法运算符用于将两个值相加起来。双目运算符的使用格式如下：

```
[variable1][operator][variable2]
```

下面是一些双目运算的例子：

```
5 + 4
3 - 2
100.4 - 92348.67
```

您将发现，大多数运算符都是双目运算符。

4.1.3 三目运算符

三目运算符是使用起来最为复杂的运算符。顾名思义，这种运算符使用三个变量。C#只有一个真正的三目运算符——条件运算符。今天的课程后面将介绍这种运算符。现在，只需知道三目运算符使用三个变量即可。

4.2 标点符号

介绍 C#的各类运算符以及具体的运算符之前，了解标点符号至关重要。标点符号是一种特殊的运算符，帮助您格式化代码，同时执行多个操作，并给编译器提供信息。您需要了解的标点符号有：

- 分号：分号的主要用途是结束每一条 C#语句，分号也可用于一些控制程序流程的 C#语句中。第 5 天的课程将介绍分号在控制语句中的用途。
- 逗号：用于将多个命令放在同一行中。在第 3 天的课程的许多例子中使用过逗号。逗号最常见的用途是用来声明同一类型的多个变量：

```
int var1, var2, var3;
```

- 圆括号：可用于多种地方。在本章后面您将看到使用圆括号来强制改变执行顺序的情况。另外，圆括号还被用于函数中。
- 花括号：花括号用于代码片段分组。前面的很多例子使用了花括号来括住类。您应该注意到了，花括号总是成对出现。

C#中标点符号的工作原理与自然语言中句子中的标点相同。例如，句子以句号或其他标点结束，而在 C#中，代码行以分号或其他标点符号结束。行是一种抽象的概念，因为在源代码清单中，一行

代码可能占用多行。正如第 2 天的课程介绍的，空白和换行符将被忽略。

注意：您将发现，也可以在例程中使用花括号，将代码分块。花括号之间的代码和花括号本身被称为代码块。

4.3 基本的赋值运算符

您首先需要了解的运算符是基本的赋值运算符，它是一个等号。在前面课程的许多例子中已经使用过该运算符。

基本赋值运算符用于赋值。例如，要将 142 赋给变量 x，可以使用下面的代码：

```
x = 142;
```

这样，编译器将把赋值运算符右边的值放到左边的变量中。请看下面的代码：

```
x = y = 123;
```

这看起来有些怪异，但却是合法的 C#代码。这里，最右边的值 123 将被放到变量 y 中，然后 y 的值将被放到变量 x 中。最后，x 和 y 的值都将为 123。

警告：不能在赋值运算符左边进行运算，例如，不能编写这样的代码：

```
1 + x = y;
```

同时，也不能将字面值或常量放在赋值运算符的左边。

4.4 数学/算术运算符

数学运算符是最常用的运算之一。在 C#中，可以执行所有的基本数学运算，包括加、减、乘、除和求模。另外，还有复合运算符，使得执行某些运算更为简洁。

注意：求模运算符也叫求余运算符（remaindering operator）。

4.4.1 加减

C#中的加法运算符用来执行加减运算。对于加法，使用加号；对于减法，则使用减号。使用这些运算符的格式如下：

```
NewVal = Val1 + Val2;
NewVal = Val1 - Val2;
```

在第一条语句中，Val2 和 Val1 相加，得到的结果被放到 NewVal 中；这种运算执行完毕后，Val1 和 Val2 的值保持不变。NewVal 中原来的值将被相加得到的结果所覆盖。

对于减法语句，则将 Val1 减去 Val2，并将得到的结果放到 NewVal 中。同样，Val1 和 Val2 的值也将保持不变，而 NewVal 中原来的值将被相减得到的结果所覆盖。

Val1 和 Val2 可以是任何数据类型的值、常量和字面值，而 NewVal 必须是变量，不过可以是与 Val1 或 Val2 相同的变量。例如，只要 Var1 是变量，则下面的语句便是合法的：

```
Var1 = Var1 - Var2
```

上述范例从 Var1 中减去 Var2，并将结果放到 Var1 中，覆盖 Var1 原来的值。下面的范例也是

合法的：

```
Var1 = Var1 - Var1;
```

这里，将 Var1 减去 Var1，由于这两个值相同，因此结果为 0。然后将结果 0 放到 Var1 中，覆盖该变量原来的值。

如果要将某个值翻番，可以使用下面的语句：

```
Var1 = Var1 + Var1;
```

Var1 与其本身相加，得到的结果被放回到 Var1 中。最终结果是，Var1 的值翻番了。

4.4.2　乘法运算符

将变量的值翻番的最简单的方法是将其乘以 2。C#中常用的乘法运算符有三个。

4.4.2.1　乘法运算

执行乘法的运算符是星号（*），它将两个值相乘，格式如下：

```
NewVal = Val1 * Val2;
```

例如，要将 Val1 的值翻番，并将得到的结果放回到 Val1 中（见最后一个关于加法的例子），可以使用下面的语句：

```
Val1 = Val1 * 2;
```

上述语句与下面的语句等效：

```
Val1 = 2 * Val1;
```

4.4.2.2　除法运算

要执行除法运算，可使用除法运算符，它是一个斜杠（/）：

```
NewVal = Val1 / Val2;
```

上述范例将 Val1 除以 Val2，并将得到的结果放到 NewVal 中。要将 2 除以 3，可使用下面的代码：

```
answer = 2 / 3;
```

4.4.2.3　求余

有时候，执行除法运算时，您只想知道余数是多少。例如，我知道 4 是 3 的一倍多，但我还可能想知道 4 除以 3 得到的余数是 1。可以使用求余（也叫求模）运算符（百分比符号%）来计算余数。例如，要计算 4 除以 3 得到的余数，可以使用下面的代码：

```
Val = 4 % 3;
```

结果是，Val 的值为 1。

请看另一个例子。有二个馅饼和 13 个人，每个馅饼都被切成 6 块，而每个人吃一块馅饼。请问还余下多少块馅饼。

要解决这种问题，首先需要知道有多少块馅饼：

```
PiecesOfPie = 3 * 6  //three pies times six pieces
```

显然，PiecesOfPie 的值为 18。现在使用求模运算符来计算余下多少块馅饼：

```
PiecesForMe = PiecesOfPie % 13;
```

上述语句将 PiecesForMe 的值设置为 18 除以 13 的余数——5。程序清单 4.1 验证了这一点。

程序清单 4.1 pie.cs：余下多少块馅饼

```
 1: // pie.cs - Using the modulus operators
 2: //---------------------------------------------
 3: class pie
 4: {
 5:    static void Main()
 6:    {
 7:       int PiecesForMe = 0;
 8:       int PiecesOfPie = 0;
 9:
10:       PiecesOfPie = 3 * 6;
11:
12:       PiecesForMe =  PiecesOfPie % 13;
13:
14:       System.Console.WriteLine("Pieces Of Pie = {0}", PiecesOfPie);
15:       System.Console.WriteLine("Pieces For Me = {0}", PiecesForMe);
16:    }
17: }
```

该程序清单的输出如下：

```
Pieces Of Pie = 18
Pieces For Me = 5
```

分析：该程序清单演示了乘法运算符和求模运算符的用法。第10行使用前面列出的公式说明了乘法运算符，而第12行则使用了求模运算符。从第14和15行打印的信息可以知道，结果同预期的完全相同。

4.4.2.4 算术赋值复合运算符

本章前面介绍过基本的赋值运算符，还有其他的赋值运算符——复合赋值运算符（如表 4.1 所示）。

表 4.1 算术赋值复合运算符

运 算 符	描 述	对应的非复合表达式
+=	x+=4	x=x+4
-=	x-=4	x=x-4
*=	x*=4	x=x*4
/=	x/=4	x=x/4
%=	x%=4	x=x%4

复合运算符提供了一种更为简洁的、执行数学运算并进行赋值的方式。如果要将某个变量的值

增加 5，可以使用下面的语句：

```
x = x + 5;
```

也可以使用复合运算符：

```
x += 5;
```

从上面可以知道，复合运算更为简洁。

提示：虽然复合运算符更简洁，但并非总是最容易理解的。使用复合运算符时，必须清晰地指出要做什么，即别忘了对代码进行注释。

4.4.3　执行单目数学运算

到目前为止，您看到的所有算符运算符都是双目的，需要两个操作数。也有很多单目运算符，只需要一个值或变量。单目算术运算符包括递增运算符（++）和递减运算符（--）。

这两个运算符分别将变量的值加 1 和减 1。下面的语句将 x 的值加 1：

```
++x;
```

其功能与下述代码等效：

```
x = x + 1;
```

而下面的语句将 x 的值减 1

```
- - x;
```

其功能与下述代码等效：

```
x = x - 1;
```

程序清单 4.2 演示了递增和递减运算符的用法。

提示：当您需要逐个遍历大量的值时，使用递增运算符和递减运算符非常方便。

程序清单 4.2　cout.cs：递增运算符和递减运算符

```
 1: // count.cs - Using the increment/decrement operators
 2: //-----------------------------------------------------
 3:
 4: class count
 5: {
 6:    static void Main()
 7:    {
 8:       int Val1 = 0;
 9:       int Val2 = 0;
10:
11:       System.Console.WriteLine("Val1 = {0}  Val2 = {1}", Val1, Val2);
12:
13:       ++Val1;
14:       --Val2;
15:
16:       System.Console.WriteLine("Val1 = {0}  Val2 = {1}", Val1, Val2);
17:
```

```
18:       ++Val1;
19:       --Val2;
20:
21:       System.Console.WriteLine("Val1 = {0}  Val2 = {1}", Val1, Val2);
22:    }
23: }
```

该程序清单的输出如下所示：

```
Val1 = 0   Val2 = 0
Val1 = 1   Val2 = -1
Val1 = 2   Val2 = -2
```

分析：该程序清单并没有完成什么壮观的工作，但说明了递增运算符和递减运算符。在第8行和第9行，两个变量被初始化为0。第11行打印这两个变量的值，以便您知道它们的值为0。第13行和第14行分别将这两个变量的值加1和减1，第16行打印它们的值，以便您能知道结果。第18～21行重复上述操作。

4.4.3.1 使用先递增和后递增运算符

递增运算符和递减运算符有独特的特征，对于新的程序员，这种特征经常会引起一些问题。假设 x 的值为 10，请看下行代码：

```
y = ++x;
```

上述语句执行后，x 和 y 的值分别是多少呢？您可能猜对了，x 和 y 的值都将是 11。现在来看下面的代码，同样假设 x 的当前值为 10：

```
y = x++;
```

上述语句执行后，x 和 y 的值又将分别是多少呢？如果您认为它们的值也将都是 11，则是不正确的。上述代码行执行后，x 的值将为 11，但 y 的值将是 10。糊涂了吧？

很简单，递增运算符可以以先递增和后递增的方式执行运算。先递增时，递增运算将首先被执行；后递增时，递增运算将最后被执行。那么，如何知道是先递增还是后递增呢？很简单，如果运算符位于变量的前面（++x），则先递增；如果位于变量的后面（x++），则后递增。递减运算符也是如此。程序清单 4.3 演示了递增运算符和递减运算符的先运算和后运算的情况。

程序清单 4.3 prepost.cs：使用单目递增运算符

```
 1:  // prepost.cs - Using pre- versus post-increment operators
 2: //---------------------------------------------------
 3:
 4: class prepost
 5: {
 6:    static void Main()
 7:    {
 8:       int Val1 = 0;
 9:       int Val2 = 0;
10:
11:       System.Console.WriteLine("Val1 = {0}  Val2 = {1}", Val1, Val2);
12:
```

```
13:       System.Console.WriteLine("Val1 (Pre) = {0}  Val2 = (Post) {1}",
14:           ++Val1, Val2++);
15:
16:       System.Console.WriteLine("Val1 (Pre) = {0}  Val2 = (Post) {1}",
17:           ++Val1, Val2++);
18:
19:       System.Console.WriteLine("Val1 (Pre) = {0}  Val2 = (Post) {1}",
20:           ++Val1, Val2++);
21:    }
22: }
```

该程序清单的输出如下：

```
Val1 = 0  Val2 = 0
Val1 (Pre) = 1  Val2 = (Post) 0
Val1 (Pre) = 2  Val2 = (Post) 1
Val1 (Pre) = 3  Val2 = (Post) 2
```

分析：理解程序清单4.3中发生的情况很重要。第8和9行同样将两个变量初始化为0。第11行打印了这些变量的值，从输出中可以知道，Val1和Val2的值都为0。第13和14行再次打印了这两个变量的值，这里打印的分别是++Val1和Val2++，即对Val1执行先递增运算，而对Val2执行后递增运算。从输出中可以知道，Val1的值被加1，然后被打印出来；而Val2首先被打印出来，然后被加1。第16～19行重复执行上述操作两次

应该	不应该
应使用复合运算符来使数学例程更为简洁	不要混淆了先递增运算符和后递增运算符。前者首先将变量的值加1，而后者最后才将变量的值加1

4.5　关系运算符

生活中总是充满问题。除了提出问题外，对事情进行比较常常也很重要。在编程中，您将对值进行比较，然后根据结果执行相应的代码。关系运算符用于对两个值进行比较。

使用关系运算符，可以确定两个值之间的关系。表 4.2 列出了关系运算符。

表 4.2　关系运算符

运 算 符	描　述
>	大于
<	小于
==	等于
!=	不等于
>=	大于等于
<=	小于等于

使用关系运算符进行比较时，将得到两个结果之一：真或假。请看下面使用关系运算符进行的比较：

- 5 < 10：5 小于 10，因此结果为真；
- 5 > 10：5 不大于 10，因此结果为假；
- 5 == 10：5 不等于 10，因此结果为假；
- 5 != 10：5 不等于 10，因此结果为真。

正如您看到的，这些结果要么为真，要么为假。知道可以检查两个值之间的关系，对于编程很重要。问题是，您如何使用这些关系？

4.5.1 If 语句

关系运算的结果可用于决策，这种决策用于改变程序的执行流程。可以结合使用 if 关键字和关系运算符来改变程序的流程。

If 关键字用于根据两个值的比较结果来决定程序流程。if 命令的标准格式如下：

```
if (val1 [operator] val2)
   statements(s);
```

其中，operator 是关系运算符，val1 和 val2 是变量、常量或字面值；而 statement(s)是单条语句或包括多条语句的代码块。代码块指的是包含在花括号中的一条或多条语句。

如果 val1 和 val2 的比较结果为真，后面的语句将被执行；否则将不被执行。图 4.1 说明了 if 语句的工作原理。

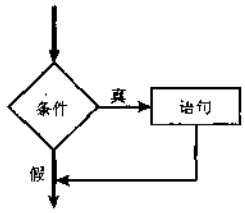


图4.1 if命令

在一个范例中使用 if 语句，可以更清楚地进行说明。程序清单 4.4 便是一个说明 if 语句用法的简单范例。

程序清单 4.4 iftest.cs：使用 if 语句

```
 1: // iftest.cs- The if statement
 2: //---------------------------------------------------
 3:
 4: class iftest
 5: {
 6:    static void Main()
 7:    {
 8:       int Val1 = 1;
 9:       int Val2 = 0;
10:
11:       System.Console.WriteLine("Getting ready to do the if...");
12:
```

```
13:       if (Val1 == Val2)
14:       {
15:          System.Console.WriteLine("If condition was true");
16:       }
17:       System.Console.WriteLine("Done with the if statement");
18:    }
19: }
```

该程序清单的输出如下：

```
Getting ready to do the if…
Done with the if statement
```

分析：该程序清单的第13行使用if语句比较两个值是否相等。如果相等，则执行第15行的打印语句；否则跳过第15行。由于在第8行和第9行给Val1和Val2指定了的值不相等，因此if条件不满足，所以不执行第15行。

将第 13 行作如下修改：

```
if (Val1 !=Val2)
```

则由于 Val1 和 Val2 确实不相等，因此 if 条件的结果为真。所以程序清单的输出将如下所示：

```
Getting ready to do the if…
If condition was true
Done with the if statement
```

警告：if语句的第一行不包括分号。例如下面的代码是不正确的：

```
if( x != x);
{
    //Statements to do when the if evaluates to true (which will never happen)
}
```

x肯定与其本身相等，因此x != x的结果将为假，花括号中的语句应该不会被执行。但由于第一行的后面有分号，这表明if语句至此便结束了，这意味着花括号中的语句将被执行。不管条件的结果是true还是false（这里为false），这些语句都将被执行。一定不要犯这样的错误，即在if语句的第一行中加上分号

4.5.2　条件逻辑运算符

现实世界常常很复杂，很多情况下，需要做多次比较来确定是否执行某个代码块。例如，您可能在某个人的年龄不小于21 岁，并且是女性时才执行某些代码。为此，需要在一条 if 语句中使用另一条 if 语句，伪代码如下：

```
if ( sex == false)
{
   if (age >=21)
   {
      // This person is a female that is 21 years old or older.
   }
}
```

有一种更简单的方式可以完成上述工作——使用条件逻辑运算符。

条件逻辑运算符让您能够使用关系运算符完成多种比较。您将使用的两个条件逻辑运算符是“和”运算符（&&）和“或”运算符（||）。

4.5.2.1 条件“和”运算符

有时候，需要验证多个条件是否都满足，前面的例子就属于这种情况。逻辑“和”运算符（&&）让您能够验证是否所有条件都满足。可以将上面的例子重新编写，如下所示：

```
if (sex == female && age >=21)
{
   // This person is a female that is 21 years old or older.
}
```

实际上，可以在 if 语句中包含两个以上的关系。不管进行的比较有多少个，仅当用&&连接起来的所有比较结果都为真时，后面的语句块才会被执行。例如在下面的范例中，仅当 if 语句中的所有三个条件都满足时，statements 代码块才会被执行。如果其中的某个条件不满足，则 statements 代码块将被跳过。

```
If ( x < 5 && y < 10 && z > 10)
{
   // statements
}
```

4.5.2.2 条件“或”运算符

有时候，并不要求所有的条件都满足，而只需要其中的一个满足即可。例如，您可能在提供的日期为星期六或星期天时，执行某些代码。在这种情况下，可以使用条件“或”运算符（||）。下面的范例使用伪代码演示了如何使用该运算符：

```
if ( day equals Sunday OR day equals Saturday)
{
   // do statements
}
```

在这个例子中，当日期为星期六或星期天时，将执行后面的语句块。仅当其中的一个条件满足时，后面的语句块便将执行。程序清单 4.5 演示了逻辑“和”运算符和逻辑“或”运算符的用法。

程序清单 4.5 and.cs：使用逻辑“和”及逻辑“或”运算符

```
 1: // and.cs- Using the conditional AND and OR
 2: //---------------------------------------------------
 3:
 4: class andclass
 5: {
 6:    static void Main()
 7:    {
 8:       int  day = 1;
 9:       char sex = 'f';
10:
11:       System.Console.WriteLine("Starting tests... (day:{0},
12:        sex:{1})", day, sex );
13:
14:       if ( day >= 1 && day <=7 )     //day from 1 to 7?
```

```
15:       {
16:          System.Console.WriteLine("Day is from 1 to 7");
17:       }
18:       if (sex == 'm' || sex == 'f' )  // Male or female?
19:       {
20:          System.Console.WriteLine("Sex is male or female.");
21:       }
22:
23:       System.Console.WriteLine("Done with the checks.");
24:    }
25: }
```

该程序清单的输出如下：

```
Starting tests…(day:1, sex:f)
Day is from 1 to 7
Sex is male or female.
Done with the checks.
```

分析：该程序清单演示了运算符 & & 和 || 的用法　第14行使用了“和”运算符（&&），要使该if语句的条件得以满足，变量day的值必须大于等于1，并小于等于7。如果day的值为1、2、3、4、5、6或7，则if条件将为真，第16行的代码被执行；否则，if条件为假，第16行将不会被执行。

第 18 行使用了“或”运算符（||） 在这里，如果变量 sex 包含的是字符 m 或 f，则第 20 行将被执行；否则被略过。

警告：请注意第18行的if条件，它检查变量包含的是不是字符m或f，这些字符是小写的，与相应的大写字母是有区别的。如果在第9行将变量sex的值设置为F或M，则第18行的if条件将不满足。

对第 8 和 9 行的代码进行修改，再运行该程序清单。得到的输出将随您设置的值而异。例如，将其修改为如下所示：

```
8:       int day = 9;
9:       char sex = 'x';
```

则程序的运行结果将如下所示：

```
Starting tests…(day:9, sex:x)
Done with the checks.
```

有时候，需要同时使用运算符 & & 和 ||。例如，您可能希望在某人年龄为不小于 21 岁，并当其为男性或女性时，执行某些代码。这可以通过同时使用运算符“和”与“或”来实现。但这样做时一定要小心。“和”运算符期望其两边的值都为 true，而“或”运算符期望其两边的值之一为 true。对于这个例子，您可能输入如下的代码来实现：

```
if ( age >= 21 && gender == male || gender == FEMALE)
   //statements
```

上述代码的运行结果将与您期望的不同。如果这个人的年龄不小于 21 岁，并为女性，则后面语句将不会执行。在这种情况下，由于&&右边的值为 false，系统不再检查后面的条件，而认为整个 if 条件为 false。要避免这种问题，可以使用括号强行改变语句的执行顺序。要得到期望的结果，可将上述代码修改为如下所示：

```
if ( age >= 21 && (gender == male || gender == FEMALE))
   //statements
```

括号最内面的代码总是首先执行。在这里，代码 gender == male || gender == FEMALE 将首先被执行，由于使用的是||，因此只要某一边的值 true，则结果便为 true。然后这部分的结果为 true，将执行&&的左边的部分，检查年龄是否不小于 21 岁。如果年龄不小于 21 岁，则后面的语句将被执行。

提示： *应使用括号来确保代码按您希望的次序执行。*

应该	不应该
应使用括号来使复杂的数学和关系运算更易于理解	不要将赋值运算符（=）和关系运算符相等（==）混淆了

4.6 逻辑按位运算符

您可能需要用到另外三个逻辑运算符：逻辑按位运算符。虽然按位运算的用法不在本书的范围之内，但本章的最后还是包含了名为“给足够勇敢者”的一节。该节对按位运算、三个逻辑按位运算符和移位运算符进行了介绍。

之所以叫按位运算符，是因为它们是针对各个位进行运算的。位是单个存储单元，其中存储了开关值（0 或 1）。本章的最后一节将介绍如何使用按位运算符对位进行操纵。

4.7 类型运算符

在本书后面开始使用类和接口后，便需要用到类型运算符。不理解接口和类，将难以完全理解类型运算符。就现在而言，您只需知道后面将使用的这种运算符，它们是：

- typeof；
- is；
- as。

4.8 sizeof 运算符

本书前面已经使用过 sizeof 运算符，它用于确定值的长度。第 3 天的课程使用过该运算符。

警告： *由于 sizeof 运算符直接操纵内存，因此应尽可能避免使用它。*

4.9 条件运算符

C#中有一个三目运算符：条件运算符。条件运算符的格式如下：

```
Condition ? if_true_statement : if_false_statement;
```

正如您看到的，两个符号将其中的三部分分开。第一部分是条件，它就像 if 语句中的条件一样，可以是结果为 true 或 false 的任何条件。

之后是一个问号，它将条件和第一条语句分开。如果条件为真，则执行第一条语句；否则执行

第二条语句。第一条语句和第二条语句之间用冒号隔开。程序清单 4.6 演示了条件运算符的用法。

条件运算符用于创建简洁的代码。如果 if 语句很简单，当其条件为 true 和 false 时，需要执行的语句都很简单，则可以使用条件运算符；否则不应使用。由于它只不过是一种简洁的 if 语句，因此应尽可能使用 if 语句本身。对于阅读您编写的代码的大多数人而言，if 语句的可读性更强，也更易于理解。

程序清单 4.6　cond.cs：使用条件运算符

```
 1: // cond.cs - The conditional operator
 2: //---------------------------------------------------
 3:
 4: class cond
 5: {
 6:    static void Main()
 7:    {
 8:       int Val1 = 1;
 9:       int Val2 = 0;
10:       int result;
11:
12:       result = (Val1 == Val2) ? 1 : 0;
13:
14:       System.Console.WriteLine("The result is {0}", result);
15:    }
16: }
```

该程序清单的输出如下：

```
The result is 0
```

分析：该程序清单很简单。在第12行，条件运算符的执行结果被赋给变量result。然后第14行打印该变量的值。条件运算符检查Val1的值是否与Val2相等。由于1不等于0，因此条件不满足。对第8行进行修改，将Val2的值设置为1，再运行该程序清单，您将发现结果为1，而不是0，这是因为1和1相等（条件已满足）。

警告：条件运算符提供了一种实现 if 语句的简短方式。虽然它更为简洁，但并非总是更容易理解。使用条件运算符时，应确保代码不会更难以理解。

4.10　运算符优先级

今天的课程介绍了大量不同的运算符。您经常需要在同一条语句中使用其中的多个运算符。在这种情况下，将出现很多问题。请看下面的语句：

```
answer = 4 * 5 + 6 / 2 - 1;
```

answer 的值将是多少呢？如果您认为是 12，则是不正确的；如果您认为是 44，则也是错误的。正确的答案是 22。

新术语：不同类型的运算符按一定的次序执行，这种次序被称为*运算符优先级*。之所以使用“优先级”，是由于某个运算符的优先级比另一些高。在上面的例子中，乘除运算的优先级高于加减运算，这意味着在执行加减运算之前，首先执行4乘以5和6除以2的运算。

表 4.3 列出了所有的运算符，位于同一行的所有运算符的优先级都相同。几乎在所有的情况下，优先级相同的运算符执行的先后次序都对结果没有影响。例如，对于 5 * 4 / 10，无论是先执行 5 乘以 4 的运算，还是先执行 4 除以 10 的运算，结果都相同。

表 4.3　　运算符优先级

优 先 级	运算符类型	运 算 符
1	主要运算符	()、.、[]、x++、x—、new、typeof、sizeof、checked、unchecked
2	单目运算符	+、–、!、++x、—x
3	乘除运算符	*、/、%
4	加减运算符	+、–
5	移位运算符	<<、>>
6	关系运算符	<、>、<=、>=、is
7	相等运算符	==、!=
8	逻辑“和”	&
9	逻辑“异或”	^
10	逻辑“或”	|
11	条件“和”	&&
12	条件“或”	||
13	条件运算符	?:
14	赋值运算符	=、*=、/=、%=、+=、–=、<<=、>>=、&=、^=、|=

4.10.1　改变优先级次序

本章前面介绍过如何使用括号来改变优先级次序。由于括号的优先级比任何运算符都高，因此其包含的任何运算符都将比外面的先执行。以上面的范例为例，可以使用括号来先执行加减运算：

```
answer = 4 * (5 + 6) / (2 - 1);
```

现在，answer 的值将是多少呢？由于括号内的运算先执行，因此编译器首先将代码解析为：

```
answer = 4 * 11 / 1;
```

最终结果为 44。括号可以嵌套，例如可以将代码写成如下所示：

```
answer = 4 * ((5 + 6) / (2 - 1));
```

编译器首先将上述代码解析为：

```
answer = 4 * (11 / 1);
```

然后再解析为：

```
answer = 4 * 11;
```

因此，answer 的最终值为 44。这里括号并没有导致不同的结果，但有时候确实会导致不同的结果。

4.11　转换数据类型

将某种类型变量的值赋给另一种类型的变量时，必须进行类型转换。另外，对两个不同数据类型的变量执行运算时，也可能需要进行类型转换。类型转换有两种：显式的和隐式的。

新术语：*隐式转换*自动进行，而不会出现错误。您在本章已经看到过很多这样的情况。但当隐式转换无法进行时，将出现什么情况呢？例如，将long变量的值赋给int变量时，将出现什么情况呢？

新术语：*显式转换*指的是强制转换数据的类型。就本章介绍的数据类型而言，最简单的显式转换方式是强制转换（cast），即强行将一种数据类型的值转换为另一种数据类型。强制转换的格式如下：

```
ToVariable = (datatype) FromVariable;
```

其中 datatype 是要将 FromVariable 转换成的数据类型。以将 long 变量转换为 int 为例，可以使用下面的语句：

```
int IntVariable = 0;
long LongVariable = 1234;
IntVariable = (int) LongVariable;
```

执行强制转换时，程序员负责确保变量能够存储转换后的值。如果目标变量无法保存转换后的值，则将进行裁剪或其他修改。很多情况下需要执行显式转换，表 4.4 列出了这些情况。

注意：就整体而言，显式转换包含隐式转换，即使在隐式转换能够进行的情况下，也可以进行强制转换。

表 4.4　必须执行显式转换的情况

原来的类型	目标类型
sbyte	byte、ushort、uint、ulong、char
byte	sbyte、char
short	sbyte、byte、ushort、uint、ulong、char
ushort	sbyte、byte、short、char
int	sbyte、byte、short、ushort、uint、ulong、char
uint	sbyte、byte、short、ushort、int、char
long	sbyte、byte、short、ushort、int、uint、ulong、char

续表

原来的类型	目 标 类 型
ulong	sbyte、byte、short、ushort、int、uint、long、char
char	sbyte、byte、short
float	sbyte、byte、short、ushort、int、uint、long、ulong、char、decimal
double	sbyte、byte、short、ushort、int、uint、long、ulong、char、float、decimal
decimal	sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double

4.12 理解操作数提升

隐式转换还与运算数提升（operand promotion）——自动将操作数从一种类型转换为另一种类型——相关。对两个变量执行基本的算术运算时，执行运算之前，它们将被转换为相同的数据类型。例如，将 byte 变量和 int 变量相加时，byte 变量将首先被提升为整数，然后执行相加运算。比 int 小的数值型变量将被提升为 int。在 int 之后，提升的次序如下：

int
uint
long
ulong
float
double
decimal

4.13 给足够勇敢者

接下来的几节是为那些足够勇敢的读者提供的，介绍了如何使用按位运算符，包括移位运算符和逻辑按位运算符。要理解这些运算符的工作原理，必须先了解变量实际上是如何存储的。

提示：理解按位运算符和内存的工作方式很有用，但对于理解C#而言并不重要。很多人将它看作是一个高级主题。

4.13.1 在内存中存储变量

要理解按位运算符，必须首先理解位。在昨天关于数据类型的课程中介绍过，存储不同的数据类型所需的位数不同。例如，char 占用两个字节，int 占用 4 个字节。另外，各种数据类型都有可存储的最大值和最小值。

一个字节为 8 位；两个字节为 16 位——8 的两倍；4 个字节为 32 位。因此，最关键是的理解位是什么。

新术语：位只不过是一个内存存储单位，就像灯泡一样，可以是开或关。在磁介质上存储信息时，位可以是正电荷或负电荷；使用光介质时，位可以是凸起或凹陷。无论是哪种情况，都是一种

状态表示0，而另一种状态表示1。

由于 1 位只能存储 1 或 0，因此其存储容量很有限。要存储大型值，必须使用一组位。例如，使用两位，可以存储 4 种不同的值：00、01、10 和 11。使用三位，则可以存储 8 种不同的值：000、001、010、011、100、101、110、111；使用 4 位，则可以存储 16 种不同的值。实际上，x 位可以存储 2^x 种不同的值，因此一个字节可以存储 2^8，即 256 种不同的值；而两个字节可以存储 65536 种不同的值。

转换这些 0 和 1 只不过是使用二进制而已。附录 D 详细介绍了如何使用二进制。就现在而言，只需知道二进制是一种计数法。

您使用十进制来计数。十进制使用 10 个数字（0–9），而二进制使用两个数字。使用十进制计数时，您使用的是 1、10、100、1000 等等。例如 13 包含 1 个 10 和 3 个 1；25 包含 2 个 10 和 5 个 1。

二进制的原理与此相同，只不过使用两个数字（0 和 1）而已。您使用的不是 10 和 100 等，而是 1、2、4、8 等等，其中的每个值都是 2 的某次方。1 为 2 的 0 次方、2 为 2 的 1 次方、4 为 2 的 2 次方等等。图 4.2 对此做了说明。

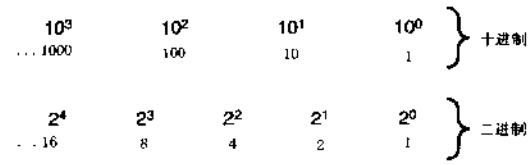


图 4.2　二进制和十进制的比较

使用二进制表示数字的原理与十进制相同。最右边一位的位置值为 1，然后依次为 2、4 等等。请看下面的数字：

```
1101
```

要将二进制数转换为十进制，可以将各位的值和位置值相乘，然后将结果全部相加。例如，最右边的一列为 1，然后依次为 0、1 和 1，因此结果为：

```
1 + (0 * 2) + (1 * 4) + (1 * 8)
```

得到的十进制数为 13。因此，二进制数 1101 对应的十进制数为 13。也可以用同样的方法将十进制数转换为二进制。随着数字的增大，所需的位数增多。为简化问题，内存实际上被划分为 8 位的单元——字节。

4.13.2　移位运算符

C#中有两种移位运算符可用于操纵位。顾名思义，这些运算符执行移位操作。使用移位运算符 >> 可以将位右移，使用 << 可以将位左移。这些运算符通过指定距离来移动变量中的位，其格式如下：

```
New_value = Value {shift-operator} number-of-positions;
```

其中，Value 是字面值或变量，shift-operator 是运算符 >> 或 << ；而 number-of-positions 是移动距离。例如，如果在一个字节的内存中存储 13，则其二进制表示为：

```
00001101
```

对其执行移位运算，可以修改其值。请看下面的例子：

```
00001101 >> 2
```

上述范例右移两位，得到的结果为：

```
00000011
```

其对应的十进制值为 3。也就是说，13>>2 等于 3。请看另一个例子：

```
00001101 << 8
```

上述范例左移 8 位，由于该值使用 1 个字节内存保存的，因此得到的结果为 0。

4.13.3 逻辑运算符

除了移位外，还可以对两个数的对应位进行组合。按位逻辑运算符有四种，如表 4.5 所示。

表 4.5 逻辑按位运算符

运 算 符	描 述
|	逻辑“或”按位运算符
&	逻辑“和”按位运算符
^	逻辑“异或”按位运算符
~	逻辑“非”按位运算符

这些运算符用于将两个二进制的对应位组合起来，得到的结果各不相同。

4.13.3.1 逻辑“或”按位运算符

使用逻辑“或”按位运算符（|）将两个值组合起来时，得到的结果如下：

- 如果两个数相应的位都是 0，则结果为 0；
- 如果至少有一个数相应的位为 1，则结果为 1。

下面是对两个 1 字节值执行逻辑“或”运算的例子：

第一个值：00001111

第二个值：11001100

结果：11001111

4.13.3.2 逻辑“与”按位运算符

使用逻辑“与”按位运算符（&）将两个值组合起来时，得到的结果如下：

- 如果两个数相应的位都是 1，则结果为 1；
- 如果至少有一个数相应的位为 0，则结果为 0。

下面是对两个 1 字节值执行逻辑“与”运算的例子：

第一个值：00001111

第二个值：11001100

结果：00001100

4.13.3.3　逻辑“异或”按位运算符

使用逻辑“异或”按位运算符（^）将两个值组合起来时，得到的结果如下：

- 如果两个数相应的位相同，则结果为 0；
- 如果两个数相应的位不同，则结果为 1。

下面是对两个 1 字节值执行逻辑“异或”运算的例子：

第一个值：00001111

第二个值：11001100

结果：11000011

程序清单 4.7 演示了这三种逻辑按位运算符的用法。

程序清单 4.7　逻辑按位运算符

```
// bitwise.cs - Using the bitwise operators
//---------------------------------------------------

class bitwise
{
   static void Main()
   {
      int ValOne = 1;
      int ValZero = 0;
      int NewVal;

      // Bitwise NOT Operator

      NewVal = ValZero ^ ValZero;
      System.Console.WriteLine("\nThe NOT Operator: \n  0 ^ 0 = {0}", NewVal);

      NewVal = ValZero ^ ValOne;
      System.Console.WriteLine("  0 ^ 1 = {0}", NewVal);

      NewVal = ValOne ^ ValZero;
      System.Console.WriteLine("  1 ^ 0 = {0}", NewVal);

      NewVal = ValOne ^ ValOne;
      System.Console.WriteLine("  1 ^ 1 = {0}", NewVal);

      // Bitwise AND Operator

      NewVal = ValZero & ValZero;
      System.Console.WriteLine("\nThe AND Operator: \n  0 & 0 = {0}", NewVal);

      NewVal = ValZero & ValOne;
      System.Console.WriteLine("  0 & 1 = {0}", NewVal);

      NewVal = ValOne & ValZero;
      System.Console.WriteLine("  1 & 0 = {0}", NewVal);

```

```
37:       NewVal = ValOne & ValOne;
38:       System.Console.WriteLine(" 1 & 1 = {0}", NewVal);
39:
40:       // Bitwise OR Operator
41:
42:       NewVal = ValZero | ValZero;
43:       System.Console.WriteLine("\nThe OR Operator: \n 0 | 0 = {0}", NewVal);
44:
45:       NewVal = ValZero | ValOne;
46:       System.Console.WriteLine(" 0 | 1 = {0}", NewVal);
47:
48:       NewVal = ValOne | ValZero;
49:       System.Console.WriteLine(" 1 | 0 = {0}", NewVal);
50:
51:       NewVal = ValOne | ValOne;
52:       System.Console.WriteLine(" 1 | 1 = {0}", NewVal);
53:    }
54: }
```

该程序清单的运行结果如下：

```
The NOT Operator:
0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0

The AND Operator:
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1

The OR Operator:
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1
```

分析：该程序清单较长，但对逻辑按位运算符进行了总结。第8行和第9行声明了两个变量，并将其值分别设置为1和0，然后使用它们来执行按位运算。每当执行完一种按位运算后，便将结果显示到控制台。查看输出可以知道，结果与前面各节描述的完全相同。

4.14 总 结

今天的课程介绍了大量关于运算符及其用法的知识。您应该知道了运算符类型（包括算术运算符、关系运算符、逻辑运算符和条件运算符）以及运算符的优先级次序。最后介绍了按位运算和按位运算符。

4.15　问与答

问：理解运算符和运算符优先级有多重要？

答：几乎每一个应用程序都要用到运算符，理解运算符优先级至关重要。如果不了解运算符的优先级，得到的结果将可能出乎您的意料。

问：今天的课程简要地介绍了二进制。理解二进制很重要吗？其他计数法是否也很重要？

答：虽然理解二进制并非生死攸关，但却很重要。在当今的计算机中，信息是以二进制格式存储的。不管表示方式是正电荷和负电荷、凸起和凹陷还是其他方式，所有的数据最终都是以二进制方式存储的　对您来说，了解二进制的原理后，理解这些实际存储的值将更容易。

除了二进制外，许多程序员还使用八进制和十六进制。八进制是一种基数为 8 的计数法；而十六进制是一种基数为 16 的计数法。附录 D 更详细地介绍了这些计数法。

4.16　作　业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

4.16.1　小测验

1. 哪个字符用于执行乘法运算？
2. 10 % 3 的结果为多少？
3. 10 + 3 * 2 的结果是多少？
4. 条件运算符是什么？
5. 哪个 C#关键字可用于改变程序的流程？
6. 单目运算符和双目运算符的区别何在？
7. 显式数据类型转换和隐式数据类型转换之间有何区别？
8. 可以将 long 值转换为 int 值吗？
9. 条件运算有哪些可能的结果？
10. 位移运算符有何功能？

4.16.2　练习

请注意，并非对所有的练习，附录 A 都提供了答案。下面的练习帮助您应用今天所学的知识。

1. 下述运算的结果是多少？

```
2 + 6 * 3 + 5 - 2 * 4
```

2. 下述运算的结果是多少？

```
4 * (8 - 3 * 2) * (0 +1) / 2
```

3. 编写一个程序，检查变量的值是否大于 65。如果是，则打印文本“The value is greater than 65!”。
4. 编写一个程序，检查一个字符变量的值是否为 t 或 T。
5. 编写一行代码，将 long 变量 MyLong 的值赋给 short 变量 MyShort。
6. 下面的程序有问题。在编辑器中输入该程序，然后编译之。哪些行会生成错误消息？是么

错误?

```
1: class exercise
2: {
3:    static void Main()
4:    {
5:       int  value = 1,
6:
7:       if ( value > 100 );
8:       {
9:         System.Console.WriteLine("Number is greater than 100");
10:      }
11:   }
12: }
```

7. 编写一行代码，将 int 变量 IntVal 的值赋给 short 变量 ShortVal。
8. 编写一行代码，将 decimal 变量 DecVal 的值赋给 long 变量 LongVal。
9. 编写一行代码，将 int 变量 ch 的值赋给字符变量 charVal。

第 5 天课程
控制语句

前面 4 天的课程介绍了大量的内容，包括如何存储信息、如何执行运算以及如何使用 if 语句来避免执行某些命令等。还介绍了一些使用 if 语句来控制流程的知识，但您常常需要对程序流程有更大的控制权。本章包含以下内容：

- 用于控制流程的其他命令；
- 使用 if 语句完成更多的任务；
- 在多个选项之间进行切换；
- 重复执行语句块多次；
- 中断代码的重复执行。

5.1　控制程序流程

通过控制程序流程，能够实现一些很有用的功能。编程时，您可能需要以许多方式来改变流程。您将需要重复执行代码块多次、跳过某个代码块或选择执行多个不同代码块中的一个。无论您以何种方式改变程序的流程，C#都有相应的选项供您完成相应的任务。改变流程的方法可以分为两类：

- 选择语句；
- 循环语句。

5.2　使用选择语句

选择语句让您能够根据条件的结果执行特定的代码块。本书前面介绍过的 if 语句就是一种选择语句，switch 语句也是。

5.2.1　再谈 if 语句

前面已经介绍过 if 语句，但有必要重温一下。请看下面的范例：

```
if( gender == 'm' || gender == 'f' )
{
    System.Console.WriteLine("The gender is valid");
}
if( gender != 'm' && gender != 'f' )
```

```
{
    System.Console.WriteLine("The gender value, {0} is not valid", gender);
}
```

本例使用一个字符变量 gender。第一个 if 语句检查 gender 是否为 m 或 f，使用了前一天课程中介绍的“或”运算符。第二个 if 语句在 gender 既不为 m 也不为 f 时，打印一条错误消息，这个 if 语句是一个确保变量的值有效的例子，如果变量的值既不为 m 也不为 f，则打印一条错误消息。

查看这两条语句，可以发现它们并不是最佳的。和其他语言一样，C#也提供了另一个用于 if 语句的关键字 else。else 关键字是专门用于 if 语句的，if...else 语句的格式如下：

```
if ( condition )
{
    // If condition is true, do these lines
}
else
{
    // If condition is false, do these lines
}
// code after if... statement
```

else 语句让您能够加入 if 语句的条件不满足时执行的代码。if 后面的语句块和 else 后面的语句块总有一个被执行，但不会都执行。其中的语句块之一执行后，程序将跳到 if...else 语句后的第 1 行代码处。

程序清单 5.1 使用 if...else 命令对上述检查 gender 值的代码做了修改。从中可以知道，其效率更高，更容易理解。

程序清单 5.1 ifelse.cs：使用 if...else 命令

```
 1: // ifelse.cs - Using the if...else statement
 2://-------------------------------------------------------
 3:
 4: class ifelse
 5: {
 6:    static void Main()
 7:    {
 8:       char gender = 'x';
 9:
10:       if( gender == 'm' || gender == 'f' )
11:       {
12:          System.Console.WriteLine("The gender is valid");
13:       }
14:       else
15:       {
16:          System.Console.WriteLine("The gender value, {0}, is not valid", gender);
17:       }
18:       System.Console.WriteLine("The if statement is now over!");
18:    }
19: }
```

该程序清单的输出如下：

```
The gender value, x, is not valid
The if statement is now over!
```

分析：第8行声明了一个char类型的变量gender，并将其值设置为x。if语句是从第10行开始的第10行检查gender的值是否为m或f。如果是，则在第12行打印一条消息，指出gender的值是有效的；否则if条件不满足，控制权转到第14行的else语句。在这个例子中，gender的值为x，因此else命令被执行，即打印一条消息，指出gender的值是无效的。然后，控制权被转到if...else语句后的第一行，即第18行。

修改第 8 行，将 gender 的值设置为 m 或 f，重新编译并执行该程序，将得到如下所示的输出：

```
The gender is valid
The if statement is now over!
```

警告：如果将gender的值设置为M或F，结果将如何呢？别忘了，C#是区分大小写的，大写字母与小写字母是不同的。

5.2.1.1　if 语句的嵌套

新术语：嵌套指的是一个语句中包含另一个语句，几乎所有的C#流程命令都可以嵌套。

要嵌套 if 语句，只需将一个 if 语句放在另一个内面即可。可以在 if 部分进行嵌套，也可以在 else 部分进行嵌套。以检查 gender 值的范例为例，可以按下面的方式使语句的效率稍高些（其中粗体部分是被嵌套的 if 语句）：

```
if( gender == 'm' )
{
    // it is a male
}
else
{
     if ( gender == 'f' )
     {
     // it is a female
     }
     else
     {
     //neither a male or a female
     }
}
```

一个 if...else 语句完全被嵌套在另一个 if 语句的 else 部分中。代码的运行方式与您预想的完全相同。如果 gender 的值不为 m，则进入到第一个 else 语句。该 else 语句中包含另一个 if 语句。该 if 语句检查 gender 的值是否为 f，如果不是，则进入相应的 else 语句。此时，您知道 gender 的值既不是 m 也不是 f，因此添加了合适的代码逻辑。

嵌套使得实现某些功能更为容易，同时您也可以堆积 if 语句。就检查 gender 值的范例而言，层叠式 if 语句是一种更佳的解决方案。

5.2.1.2　If语句的层叠

if 语句的层叠指的是将 else 与另一个 if 组合在一起。理解层叠的最佳方法是看看检查 gender 值

的范例的层叠式解决方案，如程序清单 5.2 所示。

程序清单 5.2　stacking.cs：if 语句的层叠

```
 1: // stacked.cs - Using the if...else statement
 2://-------------------------------------------------
 3:
 4: class stacked
 5: {
 6:    static void Main()
 7:    {
 8:      char gender = 'x';
 9:
10:      if( gender == 'm' )
11:      {
12:        System.Console.WriteLine("The gender is male");
13:      }
14:      else if ( gender == 'f' )
15:      {
16:        System.Console.WriteLine("The gender is female");
17:      }
18:      else
19:      {
20:        System.Console.WriteLine("The gender value, {0}, is not valid", gender);
21:      }
22:      System.Console.WriteLine("The if statement is now over!");
23:    }
24: }
```

该程序清单的输出如下：

```
The gender value, x, is not valid
The if statement is now over!
```

上述代码与嵌套 if 语句范例极其类似。主要区别在第 14 行，else 语句后面紧跟着 if，而没有花括号（语句块）。层叠式 if 语句的格式如下：

```
if ( condition 1 )
{
    // do something about condition 1
}
else if ( condition 2 )
{
    // do something about condition 2
}
else if ( condition 3 )
{
    // do something about condition 3
}
else if ( condition x )
{
     // do something about condition x
```

```
else
{
    // All previous conditions failed
}
```

上述格式较容易理解。就检查 gender 值的范例而言，只有两个条件需要检查。有时候，需要检查的条件多于两个。例如，您可以创建一个计算机程序来检查骰子的点数，并根据点数执行不同的操作。每个被层叠的 if 语句检查不同的点数（1～6），而最后的 else 语句打印错误消息，因为骰子的点数只能是 1～6。相应的代码如下：

```
if (roll == 1)
    // roll is 1
else if (roll == 2)
      // roll is 2
else if (roll == 3)
      // roll is 3
else if (roll == 4)
      // roll is 4
else if (roll == 5)
      // roll is 5
else if (roll == 6)
      // roll is 6
else
    // it isn't a number from 1 to 6
```

上述代码理解起来相对容易，因此很容易知道，对骰子 6 种可能的点数都进行了检查。如果点数不属于这 6 种情况之一，则最后的 else 语句负责处理错误逻辑或“重掷”逻辑。

注意：从上述代码可知，没有使用花括号将 if 语句括起。如果 if 或 else 内只有一条语句，则无需使用花括号将其括起。仅当有多条语句时，才需要这样做。

5.2.2 switch 语句

在根据变量的值改变程序流程方面，C#提供了一种更容易的方式：switch 语句。switch 语句的格式如下：

```
switch ( value )
{
     case result_1 :
        // do stuff for result_1
        break;
     case result_2 :
        // do stuff for result_2
        break;
     ...
     case result_n :
        // do stuff for result_x
        break;
     default:
        // do stuff for default case
        break;
```

```
}
```

从中可以知道，其中不包含 condition，而是使用了 value。value 可以是表达式，也可以是变量。value 将与每一个 case 语句中的值进行比较，直到找到匹配的值为止。如果没有找到匹配的值，则进入到 default。如果没有 default，则进入 switch 语句后的第一条语句。

找到匹配的值后，相应的 case 语句将被执行。执行完该 case 语句后，switch 语句也就执行完毕。因此，最多只会执行其中的一个 case 语句，然后将执行 switch 语句后的第一条语句。程序清单 5.3 演示了 switch 语句的用法，这里以掷骰子为例。

程序清单 5.3　roll.cs：使用 switch 语句处理掷骰子的情况

```
// roll.cs- Using the switch statement.
//---------------------------------------------------------

class roll
{
   public static void Main()
   {
      int roll = 0;

      //The next two lines set the roll to a random number from 1 to 6
      System.Random rnd = new System.Random();
      roll = (int) rnd.Next(1,7);

      System.Console.WriteLine("Starting the switch... ");

      switch (roll)
      {
         case 1:
                System.Console.WriteLine("Roll is 1");
                break;
         case 2:
                System.Console.WriteLine("Roll is 2");
                break;
         case 3:
                System.Console.WriteLine("Roll is 3");
                break;
         case 4:
                System.Console.WriteLine("Roll is 4");
                break;
         case 5:
                System.Console.WriteLine("Roll is 5");
                break;
         case 6:
                System.Console.WriteLine("Roll is 6");
                break;
         default:
                System.Console.WriteLine("Roll is not 1 through 6");
                break;
```

```
39:        }
40:        System.Console.WriteLine("The switch statement is now over!");
41:    }
42: }
```

该程序清单的输出如下所示：

```
Starting the switch...
Roll is 1
The switch statement is now over!
```

注意：您在执行上述程序时，输出的点数可能不为1。

分析：该程序清单比以前的都稍长些，但功能也强大些。首先需要注意的是第16～39行，这些代码行包含switch语句，需要重点讨论。switch语句使用存储在roll变量中的值，根据roll的值，其中的某条case语句将被执行。如果点数不为1～6，则从第39行开始的default语句将被执行。如果为1～6，则相应的case语句将被执行。

您应该注意到，每条 case 语句都以 break 命令结束，这是必须的。它指明了 case 语句的结束。如果不包含 break 命令，将出现编译错误。

为使该程序清单更有趣些，加入了第 11 和 12 行的代码，第 11 行您可能不熟悉，它创建了一个名叫 rnd 变量，该变量是一个存储随机数的对象。明天的课程将再次探讨这行代码，并详细介绍其功能。就现在而言，您只需知道它创建了一个用于存储随机数的变量即可。

第 12 行的代码也将在接下来的几天介绍，其中的命令(int) rnd.Next(1,7)生成一个 1～6 之间的随机数。

提示：可以使用第11和12行的代码来生成任何范围内的随机数，只需将其中的1和7修改为要指定的范围的上限和下限即可。第一个数是将返回的最小值，第二个数比将返回的最大值大1。例如，要生成90～100之间的随机数，可以将第12行修改为如下所示：

```
Roll = (int) rnd.Next(90, 101);
```

5.2.2.1　多种情况使用同一种解决方案

有时候，可能对于多个不同的值，都执行相同的代码。例如，对于掷骰子的情况，您可能在点数为奇数时执行某种操作，而在点数为偶数时执行另一种操作。在这种情况下，可以将多个 case 语句作为一组，相应的 switch 语句如下：

```
switch (roll)
{
    case 1:
    case 3:
    case 5:
           System.Console.WriteLine("Roll is odd");
           break;
    case 2:
    case 4:
    case 6:
           System.Console.WriteLine("Roll is even");
           break;
    default:
```

```
            System.Console.WriteLine("Roll is not 1 through 6");
            break;
    }
```

点数为 1、3 或 5 时，执行的代码相同。另外，点数为 2、4 或 6 时，执行的代码也相同。

警告：在C++中，可以通过省略break命令，来使多个case语句执行相同的代码。省略break命令，可以跳过两个case语句之间的代码。但在C#中，这样做是非法的。Case语句之间的代码不能被跳过，也就是说，如果要将case语句分组，则case语句之间不能有任何代码，而只能将代码放在每组中最后一个case语句的后面。

5.2.2.2 执行多个 case 语句

有时候，需要执行同一个 switch 语句中的多个 case 语句。在 C#中，可以使用 goto 命令来实现这种功能。可以在 switch 语句中使用 goto 命令来跳到某个 case 语句或 default 命令。下面的代码片段对上一节的 switch 语句做了修改，它使用了 goto 命令，而不是简单地跳过后面的 case 语句。

```
switch (roll)
{
    case 1:
           goto case 5;
           break;
    case 2:
           goto case 6;
           break;
    case 3:
           goto case 5;
           break;
    case 4:
           goto case 6;
           break;
    case 5:
           System.Console.WriteLine("Roll is odd");
           break;
    case 6:
           System.Console.WriteLine("Roll is even");
           break;
    default:
          System.Console.WriteLine("Roll is not 1 through 6");
          break;
}
```

虽然这个例子使用 goto 命令来演示，但采用前一节将多个 case 语句分组的方法将会简单多。您将发现，有时候 goto 命令确实能够提供您所需要的解决方案。

5.2.3 switch 语句的控制类型

只有某些数据类型能用于 switch 语句中，switch 语句的数据类型——或控制类型（governing type）——指的是 switch 语句的表达式将被解析成的类型。数据类型 sbyte、byte、short、ushort、int、uint、long、ulong、char 或文本字符串是控制类型；另一种数据类型 enum 也是合法的控制类型。数据类型 enum 将在第 8 天介绍。

如果表达式的数据类型不是上述类型，则该数据类型必有唯一的一种隐式转换方式可以将其转换为 sbyte、byte、short、ushort、int、uint、long、ulong 或字符串。如果没有这样的隐式转换方式，或者这样的方式不止一种，则编译程序时将发生错误。

注意：如果忘记了隐式转换是什么，可参阅第3天的课程。

应该	不应该
检查变量的多种不同的值时，应该使用 switch 语句	不要在 switch 或 if 语句的条件后面加上分号： if (condition);

5.3 使用循环语句

除了使用选择语句改变程序流程外，有时候可能需要重复执行某个代码片段多次。为了能够重复执行代码，C#提供了多种循环语句。循环语句可以执行代码块 0 或更多次。每执行一次代码，便是一次循环。

C#中的循环语句包括：

- while；
- do；
- for；
- foreach。

5.3.1 while 语句

while 命令用于条件为真时，不断重复执行代码块，其格式如下：

```
while (condition)
   statement(s)
```

图 5.1 对此做了说明。

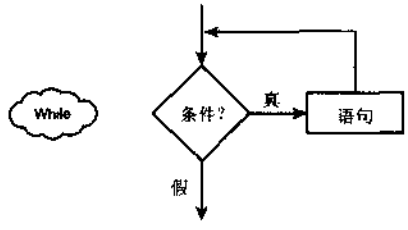


图 5.1　while 命令

从上图可以知道，while 语句使用了一个条件语句，如果条件满足，则语句将被执行，否则不执行，程序流程进入 while 语句后面的语句。程序清单 5.4 使用 while 语句打印 10 个 1～10 之间的随机数及其平均值。

程序清单 5.4　average.cs：使用 while 命令

```
1: // average.cs Using the while statement.
2: // print the average of 10 random numbers that are from 1 to 10.
3: //---------------------------------------------------------------
```

```
 4:
 5: class average
 6: {
 7:    public static void Main()
 8:    {
 9:       int ttl = 0;  // variable to store the running total
10:        int nbr = 0;  // variable for individual numbers
11:       int ctr = 0;  // counter
12:
13:       System.Random rnd = new System.Random();  // random number
14:
15:       while ( ctr < 10 )
16:       {
17:          //Get random number
18:          nbr = (int) rnd.Next(1,11);
19:
20:          System.Console.WriteLine("Number {0} is {1}", (ctr + 1), nbr);
21:
22:          ttl += nbr;       //add nbr to total
23:          ctr++;           //increment counter
24:       }
25:
26:       System.Console.WriteLine("\nThe total of the {0} numbers is {1}", ctr, ttl);
27:       System.Console.WriteLine("\nThe average of the numbers is {0}", ttl/ctr );
28:    }
29: }
```

注意：您执行该程序时，得到的输出将与这里的不同。因为这里使用的是随机数，这些数字在每次执行程序时都不同。

该程序清单的输出如下：

```
Number 1 is 2
Number 2 is 5
Number 3 is 4
Number 4 is 1
Number 4 is 1
Number 6 is 5
Number 7 is 2
Number 8 is 5
Number 9 is 10
Number 10 is 2

The total of the 10 numbers is 37

The average of the numbers is 3
```

分析：这里使用了今天介绍的生成随机数的代码，只不过生成的随机数不在1～6之间，而是1～10之间。这是在第18行完成的，这里将随机数的范围扩大到10。在以这种方式使用随机变量之前，第13行对其进行了初始化。

while 语句是从第 15 行开始的。该 while 语句的条件是检查计数器是否小于 10。由于该计数器在第 11 行被初始化为 0，因此条件是满足的，所以 while 内的语句将被执行。while 语句取得一个 1 ~ 10 之间的随机数（第 18 行），并将其加入到累计计数器 ttl 中（第 22 行）。第 23 行将计数器变量 ctr 加 1，然后便是 while 语句的结尾处（24 行）。程序流程将自动回到第 15 行的 while 条件处，对条件重新进行评估，看其是否仍然满足。如果满足，则再次执行语句。这一过程将一直持续下去，直到 while 条件不满足为止。就这个程序而言，当 ctr 的值为 10 时，条件便不满足了。此时程序流程将进入第 25 行，即紧跟在 while 语句后的一行。

While 语句后面的代码打印 10 个随机数的总和及平均值。然后结束程序。

警告： 为确保while语句最终得以结束，必须在语句中修改某些将影响条件的因素。如果条件永远不会不满足，则while语句将成为死循环。还有另一种终止while循环的方法：使用break语句，这将在下一节进行介绍。

5.3.1.1　中断和继续 while 语句

可以在条件不满足之前中断 while 语句，也可以在执行完毕所有的语句之前中断 while 语句的一次循环。

要中断 while 语句，提早结束它，可以使用 break 命令。Break 语句将使程序流程立刻转到 while 语句后面的第一条语句。

也可以使 while 语句立刻进入到下一次循环，这是通过 continue 语句实现的。Continue 语句导致程序流程进入到 while 的条件语句。程序清单 5.5 演示如何在 while 语句中使用 break 和 continue 语句。

程序清单 5.5　even.cs：使用 break 和 continue 语句

```
 1: // even.cs- Using the while with the break and continue commands.
 2: //-------------------------------------------------------------------------
 3:
 4: class even
 5: {
 6:    public static void Main()
 7:    {
 8:       int ctr = 0;
 9:
10:       while (true)
11:       {
12:          ctr++;
13:
14:          if (ctr > 10 )
15:          {
16:             break;
17:          }
18:          else if ( (ctr % 2) == 1 )
19:          {
20:             continue;
21:          }
22:          else
23:          {
24:             System.Console.WriteLine("...{0}...", ctr);
25:          }
```

```
26:        }
27:        System.Console.WriteLine("Done!");
28:    }
29: }
```

该程序清单的输出如下所示：

```
...2...
...4...
...6...
...8...
...10...
Done!
```

分析：这个程序清单打印偶数，而忽略奇数。当计数器的值大于10后，通过break语句结束while语句。

第 8 行声明了一个计数器变量 ctr，并将其值设置为 0。从第 10 行开始为 while 语句，由于使用 break 语句来结束循环，因此第 10 行的条件被简单地设置为真。这实际上创建了一个死循环，因此需要使用 break 语句来结束 while 语句的循环。在 while 语句中，首先将 ctr 的值加 1，如第 12 行所示。然后检查 ctr 的值是否大于 10，如第 14 行所示。如果 ctr 大于 10，则执行第 16 行的 break 语句，终止 while 循环，进入第 27 行。

如果 ctr 小于 10，则执行第 18 行的 else 语句。该 else 语句与一个 if 语句层叠起来，后者检查当前的数字是否为奇数，这是使用求模运算符实现的。如果计数器为偶数，则其除以 2 的余数将为 0，如果为奇数，则余数将为 1。当计数器为奇数时，将执行第 20 行的 continue 语句，即回到 while 语句的开始位置，重新检查条件。由于条件总是为真，因此将再次执行 while 语句。这将首先将计数器加 1，如第 12 行所示，然后执行其他检查。

如果计数器不是奇数，则第 22 行的 else 语句将被执行。该 else 语句只包含一条语句，它调用 WriteLine，打印计数器的值。

5.3.2 do 语句

如果首次检查时，while 语句的条件便不满足，则 while 语句一次也不会执行。有时候，希望语句至少执行一次。在这种情况下，do 语句将是一种更好的解决方案。

do 语句的格式如下：

```
do
   statement(s)
while (condition);
```

图 5.2 说明了这种格式。

从上图可以知道，do 语句首先执行其包含的语句，然后对 while 语句中的条件进行检查。这里的 while 语句及其条件的运行方式与前面介绍的 while 语句完全相同。如果条件为真，则返回到前面的语句；否则，进入 do...while 语句后的第一行。程序清单 5.6 演示了 do 语句的用法。

注意：由于do语句中要使用while语句，因此do语句常被称为do...while语句。

程序清单 5.6 do_it.cs：使用 do 命令

```
1: // do_it.cs Using the do statement.
```

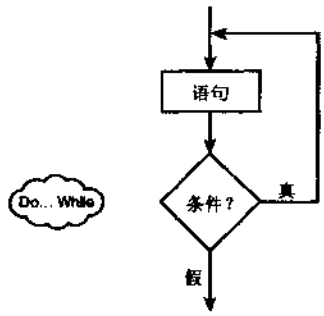


图 5.2　do 命令

```
 2: // Get random numbers (from 1 to 10) until a 5 is reached.
 3: //-------------------------------------------------------------
 4:
 5: class do_it
 6: {
 7:    public static void Main()
 8:    {
 9:       int ttl = 0;  // variable to store the running total
10:       int nbr = 0;  // variable for individual numbers
11:       int ctr = 0;  // counter
12:
13:       System.Random rnd = new System.Random();  // random number
14:
15:       do
16:       {
17:          //Get random number
18:          nbr = (int) rnd.Next(1,11);
19:
20:          ctr++;           //number of numbers counted
21:          ttl += nbr;       //add nbr to total of numbers
22:
23:          System.Console.WriteLine("Number {0} is {1}", ctr, nbr);
24:
25:       } while ( nbr != 5 );
26:
27:       System.Console.WriteLine("\n{0} numbers were read", ctr);
28:       System.Console.WriteLine("The total of the numbers is {0}", ttl);
29:       System.Console.WriteLine("The average of the numbers is {0}", ttl/ctr);
30:    }
31: }
```

该程序清单的输出如下所示：

```
Number 1 is 1
Number 2 is 6
Number 3 is 5

3 numbers were read
```

```
The total of the numbers is 12
The average of the numbers is 4
```

由于该程序清单使用的是随机数，因此您执行该程序得到的结果很可能不同于这里显示的。您将得到一系列的数字，但最后一个为 5。

在该程序中，您希望至少执行操作（获取一个随机数）一次。然后一直执行这种操作，直到条件（获取的随机数为 5）满足为止。这种情况非常适合使用 do 语句。该程序清单与程序清单 5.4 极其类似。第 9～11 行声明了几个变量来记录总数和计数器。第 13 行声明了一个用于获取随机数的变量。

do 语句是从第 15 行开始的。do 语句体（第 16～24 行）被执行。首先获取下一个随机数，同样这里将 1～10 之间的随机数赋给变量 nbr。第 20 行在每读取一个随机数后将 ctr 变量的值加 1，来记录读取了多少个随机数。第 21 行将取得的随机数加入到累积值中。记住下面的代码：

```
ttl += nbr;
```

与下述代码是等效的：

```
ttl = ttl + nbr;
```

第 23 行将取得的随机数及其编号打印到屏幕上。

第 25 行是 do 语句的条件部分。这里的条件是 nbr 不等于 5。只要取得的随机数 nbr 不等于 5，do 语句体便继续执行。当取得 5 时，循环将终止。从程序的输出可以知道，其中总是只有一个“5”，并且是最后一个数。

第 27～29 行打印关于取得的随机数的统计信息。

5.3.3 for 语句

虽然 do...while 和 while 语句提供了控制代码循环所需的所有功能，但还有其他的语句也可用于控制代码的循环。介绍 for 语句之前，请先看一下下面的代码片段：

```
ctr = 1
while (ctr < 10)
{
   //do some stuff
   ctr ++;
}
```

从中可以知道，使用了一个计数器来循环执行 while 语句。其流程如下：

1. 将计数器的值设置为 1。
2. 检查计数器是否小于 10。如果不小于 10（条件不满足），则结束循环。
3. 执行某些操作。
4. 将计数器加 1。
5. 回到第 2 步。

在循环中，这些步骤很通用。由于这些步骤是通用的，因此 for 语句也使用这些步骤，不过格式更为简单：

```
for (initializer; condition; incrementor)
   statement(s)
```

从上述 for 语句的格式可以知道，括号内包含三部分：initializer、condition 和 incrementor，各部分之间用分号隔开。省略其中的某部分时，分号仍需保留。

for 语句执行时，首先执行 initializer，它只在开始时执行一次，以后再也不执行。

执行 initilizer 后，对 condition 进行评估。和 while 语句中的条件一样，这里的条件也只能为真或假。如果为真，则 statement(s)将被执行。

语句或语句块执行后，将返回到 for 语句的开始位置，执行 incrementor。Incrementor 可以是任何合法的 C#表达式，但通常用于增加计数器的值。

Incrementor 被执行后，将再次对条件进行评估。只要条件仍为真，语句块便会被执行，然后执行 incrementor。这一过程将一直重复下去，直到条件为假为止。图 5.3 说明了 for 语句的流程。

介绍程序清单之前，我们来将本节开始的 while 语句修改为 for 语句：

```
for ( ctr = 1; ctr < 10; ctr++)
{
   //do some stuff
}
```

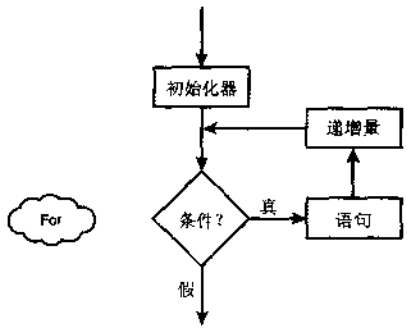


图 5.3　for 命令

上述 for 语句比前面使用 while 的代码简单得多。

该 for 语句的执行步骤如下：

1. 将计数器的值设置为 1。
2. 检查计数器是否小于 10。如果不小于 10（条件不满足），则结束 for 语句。
3. 执行某些操作。
4. 将计数器加 1。
5. 回到第 2 步。

其步骤与前面的 while 语句完全相同，区别在于 for 更简洁，更易于理解。程序清单 5.7 演示了一个更强大的 for 语句。实际上，该程序清单的功能与程序清单 5.4 完全相同，只不过更为简洁而已。

程序清单 5.7　foravg.cs：使用 for 语句

```
1: // foravg.cs Using the for statement.
2: // print the average of 10 random numbers that are from 1 to 10.
3: //---------------------------------------------------------------
4:
5: class average
```

```
 6: {
 7:    public static void Main()
 8:    {
 9:       int ttl = 0;  // variable to store the running total
10:       int nbr = 0;  // variable for individual numbers
11:       int ctr = 0;  // counter
12:
13:       System.Random rnd = new System.Random();  // random number
14:
15:       for ( ctr = 1; ctr <= 10; ctr++ )
16:       {
17:          //Get random number
18:          nbr = (int) rnd.Next(1,11);
19:
20:          System.Console.WriteLine("Number {0} is {1}", (ctr), nbr);
21:
22:          ttl += nbr;       //add nbr to total
23:       }
24:
25:       System.Console.WriteLine("\nThe total of the 10 numbers is {0}", ttl);
26:       System.Console.WriteLine("\nThe average of the numbers is {0}", ttl/10 );
27:    }
28: }
```

该程序清单的输出如下：

```
Number 1 is 10
Number 2 is 3
Number 3 is 6
Number 4 is 5
Number 4 is 7
Number 6 is 8
Number 7 is 7
Number 8 is 1
Number 9 is 4
Number 10 is 3

The total of the 10 numbers is 54

The average of the numbers is 5
```

分析：该程序清单的大部分内容都与程序清单5.4相同。但也有一些区别。第15行使用了for语句。计数器被初始化为1，这使在第20行使用WriteLine例程来显示值更为容易。另外，for语句中的条件语句也做了相应的调整。

程序执行到 for 语句处时，将出现什么情况呢？简单地说，计数器被设置为 1，然后验证条件。这时候，计数器小于等于 10，因此 for 语句体被执行。for 语句体（第 16～23 行）执行完毕后，回到第 15 行中 for 语句的 incrementor 处。这里将计数器的值加 1。然后再次对条件进行检查，如果为真，则再次执行 for 语句体。这一过程将一直重复下去，直到条件为假为止。在这个程序中，当计数器

的值为 11 时，条件便为假。

5.3.3.1　for 语句表达式

可以在 initilizer、condition 和 incrementor 中完成很多工作。可以将表达式放在这些地方，甚至可以放置多个表达式。

如果在 for 语句的某部分中使用多个表达式，则需要分开它们。分隔符是逗号。作为一个例子，下面的 for 语句初始化两个变量，并递增它们的值。

```
for ( x = 1, y = 1; x + y < 100; x++, y++)
   //Do something...
```

除了可以包含多个表达式外，还可以省略某些部分。在下面的范例中，所有的工作都是在 for 语句的控制结构中完成的，for 语句体只有一条空语句——一个分号。

```
for (x = 0; ++x <= 10; System.Consol.WriteLine("{0}", x))
```

上述简单的代码行实际上完成了很多任务，它打印数字 1～10。本章后面的一个练习要求您将其改写为一个完整的程序清单。

警告：不要在for语句的控制结构中完成太多的工作，要确保它不要过于复杂，以至难以理解。

5.3.4　foreach 语句

foreach 语句以类似于 for 语句的方式执行循环，但有一种特殊用途，即可用于遍历诸如数组等集合。foreach 语句、集合和数组将在第 8 天的课程中介绍。

5.3.5　再谈 break 和 continue 语句

在前面关于 while 语句的一节中已经介绍过 break 和 continue，另外还在 switch 语句中使用过 break 命令。这两个命令也可以用于其他的程序流程语句中。

在 do...while 语句中，break 和 continue 的运行方式与在 while 语句中完全相同。continue 命令返回到条件语句处；而 break 命令转到 do...while 后面的语句。

在 for 语句中，continue 语句转到 incremetor 语句处，然后对条件进行检查，如果为真，则继续执行 for 语句。break 语句转到 for 语句之后的语句。

break 命令退出当前循环；而 continue 命令开始下一次循环。

5.4　使用 goto

不管您使用的是哪种编程语言，goto 语句都充满争议。由于 goto 语句能够无条件地改变程序流程，因此非常强大。但权力和义务总是并存的。许多开发人员避免使用 goto 语句，因为它容易导致代码难以理解。

goto 语句的使用方式有三种。在前面您已经看到了，在 switch 语句中，可以以两种方式使用 goto 语句：goto case 和 goto default。在前面讨论 switch 语句时，以这两种方式使用过 goto 语句。

第三种 goto 语句的使用格式如下：

```
goto label;
```

使用这种格式的 goto 语句，可以转到一条标签语句。

5.4.1 标签语句

标签语句只不过是一个指示位置的命令。标签的格式如下：

```
label_name:
```

注意，它后面跟的是冒号，而不是分号。程序清单 5.8 演示了如何使用带标签的 goto 语句。

程序清单 5.8 Score.cs：使用带标签的 goto 语句

```
 1: // score.cs   Using the goto and label statements.
 2: // Disclaimer: This program shows the use of goto and label
 3: //           This is not a good use; however, it illustrates
 4: //           the functionality of these keywords.
 5: //-----------------------------------------------------------
 6:
 7: class score
 8: {
 9:    public static void Main()
10:    {
11:       int score = 0;
12:       int ctr = 0;
13:
14:       System.Random rnd = new System.Random();
15:
16:       Start:
17:
18:       ctr++;
19:
20:       if (ctr > 10)
21:          goto EndThis;
22:       else
23:          score = (int) rnd.Next(60, 101);
24:
25:       System.Console.WriteLine("{0} - You received a score of {1}",
26:                 ctr, score);
27:
28:       goto Start;
29:
30:      EndThis:
31:
32:       System.Console.WriteLine("Done with scores!");
33:    }
34: }
```

该程序清单的输出如下所示：

```
1 - You received a score of 83
2 - You received a score of 99
3 - You received a score of 72
4 - You received a score of 67
5 - You received a score of 80
6 - You received a score of 98
```

```
7 - You received a score of 64
8 - You received a score of 91
9 - You received a score of 79
10 - You received a score of 76
Done with scores!
```

分析：该程序清单的目的相对简单，它通过取得10个60～100之间的随机数来打印10个分数。这里使用随机数的方式与前面介绍的相同，只是做了细微的修改。在第23行，指定的随机数下限为60，而不是1；另外，由于要获取60～100的随机数，因此上限被设置为101。通过将101设置为上限，取得的随机数将小于101。

该程序清单的重点在第 16、21、28 和 30 行。第 16 行包含一个名为 Start 的标签，由于它是一个标签，因此程序执行时将略过这一行，进入第 18 行，将计数器的值加 1。然后，检查第 20 行的 if 语句中的条件。如果计数器的值大于 10，则第 21 行的 goto 语句被执行，即跳到第 30 行的标签 EndThis 处。由于此时计数器不大于 10，因此第 22 行的 else 语句被执行。该 else 语句取得一个随机的分数（如第 23 行所示，这在前面已经介绍过了）。第 25 行打印取得的分数，然后进入到第 28 行，该行无条件地跳到 Start 标签处。由于 Start 标签位于第 16 行，因此将跳到第 16 行。

该程序清单完成的循环与 while、do 或 for 语句类似。在很多情况下，可以使用其他编程元素来代替 goto 语句。如果有其他的方法可供选择，则应该使用这些方法。

提示：应尽可能避免使用goto语句，它会导致被称为意大利面条式代码（spaghetti code）的情形。意大利面条式代码指的是所有代码纠缠在一起，因此难以知道代码从哪里开始，到哪里结束。

5.5　程序流程命令的嵌套

今天介绍的所有程序流程命令都可以被嵌套。将程序流程命令嵌套时，应确保命令正确结束。如果嵌套不正确，可能导致逻辑错误，甚至语法错误。

应该	不应该
一定要对代码进行注释，以清楚地说明程序的功能和流程	除非绝对必要，否则不要使用 goto 语句

5.6　总　结

今天介绍的内容很多，而这些知识几乎在所有的 C#应用程序中都要用到。

今天的课程再次对 C#语句的一些基本结构进行了介绍。首先通过介绍 else 语句，扩展了您关于 if 语句的知识。今天还介绍了另一种选择语句：switch 语句。接着讨论了控制程序循环流程的语句，包括 while、do 和 for 语句。另外，还简要介绍了另一个命令 foreach，该命令将在第 8 天详细进行介绍。您除了学会了如何使用命令外，还知道这些命令可以嵌套。最后，介绍了 goto 语句以及如何将其与 case、default 和标签结合起来使用。

5.7　问与答

问：还有其他类型的控制语句吗？

答：有 throw、try、catch 和 finally。后面的课程将介绍这些语句。

问：可以在 switch 语句中使用文本字符串吗？

答：可以。字符串是 switch 语句的一种“控制类型”，这意味着可以在 switch 语句中使用字符串变量，进而在 case 语句中使用字符串值。记住，字符串只不过是用双引号标示的文本而已。在后面的一个练习中，您将创建一条使用字符串的 switch 语句。

问：为何人们对 goto 语句的评价如此差？

答：goto 语句臭名昭著。如果以结构化、有组织的方式偶尔使用，goto 语句可以帮助解决一些编程方面的问题，goto case 和 goto default 就是绝佳的例子。goto 语句之所以臭名昭著，主要在于它经常被滥用。程序员以非结构化的方式使用它，快速地从代码的一个地方跳到另一个地方。在面向对象的编程语言中，程序的结构化程度越高，程序越好，维护起来越容易。

5.8 作 业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

5.8.1 小测验

1. 为重复执行代码行多次，C#提供了哪些命令？
2. while 语句中的语句将至少执行多少次？
3. do 语句中的语句将至少执行多少次？
4. 下面的 for 语句中，哪一部分是条件语句？

```
for ( x = 1; x == 1; x++)
```

5. 上述 for 语句的 incrementor 语句是什么？
6. 什么语句用于结束 switch 语句中的 case 表达式？
7. 标签中使用的是什么标点？
8. 在 for 语句中，使用哪个标点符号来分隔表达式？
9. 嵌套指的是什么？
10. 什么命令用于跳到下一次循环。

5.8.2 练习

1. 编写一条 if 语句，检查变量 file-type 的值是 s、m 还是 j，并根据 file-type 的值打印下面的消息：

```
s: The filer is single
m: The filer is married filing at the single rate
j: The filer is married filing at the joint rate
```

2. 下面的 if 语句合法吗？这些代码执行后，x 的值将为多少？

```
int x = 2;
int y = 3;
if( x == 2) if (y > 3) x=5; else x =9;
```

3. 编写一个从 99 数到 1 的 while 语句。

4. 将练习 3 的 while 语句改为 for 语句。

5. 下面的程序清单正确吗？如果正确，其功能是什么？如果不正确，错在哪里？

```
// Ex5-5.cs, Exercise 5 for Day 5
//-----------------------------------------------------------------------

class score
{
    public static void Main()
    {
     int score = 99;

     if ( score == 100 );
     {
       System.Console.WriteLine("You got a perfect score!");

     }
     else
       System.Console.WriteLine("Bummer, you were not perfect!");
    }
}
```

6. 编写一个这样的 for 循环，即在其 initializer、condition 和 incrementor 部分完成打印数字 1～10 的工作，for 语句体是一条空语句。

7. 编写一条根据变量 name 的值执行不同操作的 switch 语句。如果该变量的值为 Robert，则打印消息 "Hi Bob"；如果为 Richard，则打印消息 "Hi Rich"；如果为 Barbara，则打印消息 "Hi Barb"；如果为 Kalee，则打印 "You Go Girl!"。对于其他情况，则打印 "Hi x"，其中 x 为变量 name 的值。

8. 编写一个掷骰子 100 次的程序，并打印出各种点数的出现次数。

第 6 天课程
类

正如第 2 天介绍的，对于面向对象语言而言，类至关重要，对于 C#来说也是如此。在本书前面介绍的每一个程序中都使用了类。今天将介绍以下内容：

- 面向对象编程涉及的概念；
- 如何声明类；
- 如何定义类；
- 类成员；
- 创建自己的数据成员；
- 在类中实现属性；
- 名称空间。

6.1 再谈面向对象编程

第 2 天介绍过，C#是一种面向对象语言。要充分利用 C#，必须理解面向对象语言的概念。接下来的几节将简要地重温一下第 2 天介绍的一些概念，然后介绍如何将这些概念用于实际的 C#程序中。

第 2 天介绍过，面向对象语言的关键特征包括：

- 封装；
- 多态；
- 继承；
- 重用。

6.1.1 封装

封装指的是建立类（包），其中包含所需的所有东西。在面向对象编程中，这意味着您可以创建一个类，其中存储了您需要的所有变量以及操纵这些数据的例程。可以创建一个 circle 类，来存储关于圆的信息。这可能包括圆心的位置、半径以及对所有的圆都通用的例程。这些例程可能包括计算圆面积和周长、改变圆心位置和半径。

通过封装圆，可以让用户无需关心圆的工作原理，而只要知道如何与其交互即可。它将圆的内部工作方式隔离了开来，即类中的变量可以被修改，但对用户而言，是不可见的。例如，可以存储

圆的直径，而不是半径。如果封装了功能和数据，则上述修改只会影响类。对于使用该类的任何程序，都无需进行修改。今天和明天的课程中都包括直接使用 Circle 类的程序。

注意：封装常被称为“黑盒化（black boxing）”　黑盒化指的是隐藏功能或内部处理方式。对于圆而言，传入半径可以获得其面积。您无需关心这是如何得到的，只要知道得到的答案是正确的。

6.1.2　多态

多态指的是能够呈现多种形式的功能，即程序能够使用您发送给它的东西进行工作。例如，在前几天使用过的 WriteLine()中，您可以使用{0}来创建参数字段。该字段打印什么值呢？它可以打印任何类型的变量或字符串。至于如何打印它们，则是由 WriteLine( )例程负责的。该例程能够接受您传递给它的大部分数据类型，因此是多态的。

以圆为例，您可能想调用圆对象来取得其面积。为此，可以传递圆周上的三个点或圆心和半径。无论采用哪种方式，得到的结果都是相同的。另外，圆是一种形状，因此圆的一种多态特征是能够理解这种形状，并以这种形状的方式做出反应。

6.1.3　继承

正如第 2 天介绍过的，继承是最复杂的面向对象概念。继承在一个类（对象）是另一个类的扩展时发生。

在许多关于面向对象编程的书籍中，使用动物方面的类比来说明继承。这种类比首先使用了动物是一种生物的概念。

爬行动物属于动物，但是变温的。爬行动物具备动物的所有特征，但同时有自己特有的特征。而蛇属于爬行动物，但修长且没有脚。它具备爬行动物的所有特征，但同时有自己特有的特征。可以这么说，蛇继承了爬行动物的特征，而爬行动物继承了动物的特征。

在第 11 天的课程中，您将知道如何将这种概念用于类和编程中。

6.1.4　重用

创建类后，可以重复使用它来创建大量的对象。通过使用继承以及前面介绍的其他一些特性，可以创建出能够在许多程序中以许多不同的方式重复使用的例程。通过封装功能，可以创建出经过测试并被证明能够正确运行的例程。用户无需详细测试功能，而只要正确使用它即可。这使得重用这些例程简单而快速。

6.1.5　对象和类

第 2 天使用蛋糕刀和蛋糕来说明了类和对象。现在有了这些概念后，可以使用它们来编写一些代码了。

注意：接下来的三天中，将介绍关于类的知识，首先介绍一些非常简单的范例，然后对其进行扩展。

6.2　定义类

为简化起见，关键字 class 被用来定义类。类的基本结构如下：

```
class identifier
{
   class-body;
```

```
}
```

其中 identifier 是给类指定的名称，class-body 是组成类的代码。

类名和其他变量名一样，可以指定。应该给类指定一个有意义的名称，即对类的功能进行描述。

Microsoft .NET 框架中包含大量内置类，本书一开始便使用了其中的一个：Console 类。Console 类包含多个数据成员和例程，前面已经使用过其中的几个例程，包括 Write 和 WriteLine。这个类的名称（标识符）为 Console。Console 的类体中包含了例程 Write 和 WriteLine 的代码。阅读完明天的课程后，您便能够创建并命名自己的、包含与 Console 类中类似的例程的类。

6.3 类声明

定义类后，便可以使用它来创建对象。类只是一种用于创建对象的定义。类本身并不能存储信息或执行例程，而只是用于声明对象。对象可以按类定义的方式来存储信息和执行例程。

注意： *声明对象常被称为实例化。换句话说，对象是类的一个实例。*

使用类来声明对象的格式如下：

```
class_name object_identifier = new class_name( );
```

其中 class_name 是类的名称，而 object_identifier 是要声明的对象的名称。例如，如果有一个名为 point 的类，则可以使用下面的代码来创建一个名为 startingPoint 的对象：

```
point startingPoint = new point( );
```

类名为 point，声明的对象的名称为 startingPoint。由于 startingPoint 是一个对象，因此它包含 point 类声明的数据和例程。

您可能会问，上述声明代码行中的其他项是干什么的？这里最重要的是以前未介绍过的关键字 new。

顾名思义，关键字 new 用于创建一个新东西，这里创建一个新的点。由于 point 是一个类，因此创建的是一个对象。关键字 new 指示要创建一个新的实例，这里创建的新实例是一个 point 对象。

使用类来声明对象时，必须在赋值运算符右边的类名后面加上圆括号，这样可以使用类来构造一个新对象。

警告： *不加上构造代码new classname，将声明一个类，但编译器不会构造其内部结构。一定要将new classname赋给声明的对象名，以确保所有的东西都将构造好。明天将更详细地介绍这种初始化结构。*

再来看一下上述语句：

```
point startingPoint = new point( );
```

下面分别介绍其中各部分的功能。

```
point startingPoint
```

使用 point 类来声明一个名为 startingPoint 的对象。上述代码与声明诸如 int 和 decimal 等其他数据类型类似。

```
startingPoint =
```

将赋值运算符右边的结果赋给左边的变量。这里变量是一个对象，一个名称为 startingPoint、类型为 point 的对象。

```
new point( )
```

这部分代码构造 point 对象。带圆括号的类名表示要构造（创建）一个这种类的对象。New 指示为该新对象预留一些内存空间。记住，类只是一种定义，它不存储任何东西。对象需要存储信息，因此需要为它预留内存。关键字 new 用于预留内存。

和所有的语句一样，该声明以分号结束，分号指示语句结束。

6.3.1 类成员

知道类的整体结构以及如何使用类来创建对象后，我们来看一下类中可以包含哪些东西。类体中可包含的东西主要有两种：数据成员和函数成员。

数据成员包括变量和常量，这包括第 3 天介绍的任何数据类型的变量以及后面将介绍的高级数据类型的变量。数据成员甚至可以是其他类。

类体中的另一种元素是函数成员。函数成员指的是执行操作的例程。这种操作可以简单到取得一个值，也可以复杂到使用不同数目的值，将一行文本写到屏幕上，就像 Write 和 WriteLine 那样。Write 和 WriteLine 是 Console 类的成员函数。明天将介绍如何创建和使用自己的成员函数。接下来我们来介绍数据成员。

6.4 数据成员（字段）

新术语：*变量也被称为字段。正如前面指出的，类中的数据成员是一个属于类成员的变量。*

对于前面介绍的 point 类，您希望它包含用于存储点的 x 坐标和 y 坐标的数据成员。这些坐标值可以是很多种数据类型中的一种，如果是 int 型，则可以使用下面的方式来定义 point 类：

```
class point
{
   int x;
   int y;
}
```

这实际上就是一个非常简单的 point 类的代码。现在需要加入另一个元素，即访问限定符 public。不加入单词 public，将无法在 point 的外面访问 x 和 y。除非特别说明，否则只能在声明变量的代码块内访问该变量。这里的语句块是 point 类的定义。

注意：*代码块是位于花括号内的代码片段，类体就是一个代码块。*

对 point 类所做的修改比较简单，加入访问限定符 public 后，point 类将变成如下所示：

```
class point
{
   public int x;
   public int y;
}
```

虽然 point 类包含的是两个整数，但您可以在该类中使用任何数据类型。例如，您可以创建一个名为 FullName 的类，它包含三个字符串，分别存储姓、名和中名。也可以创建一个名为 Address

类，它包含 FullName 类和用于存储地址的其他字符串。还可以创建一个 customer 类，它包含一个用于存储客户编号的 long 变量、一个 Address 类、一个用于存储账户结余的 decimal 变量和一个指定该账户是否有效的布尔型变量等。

6.4.1 访问数据成员

声明数据成员后，需要取得它们的值。正如前面介绍的，访问限定符 public 让您能够从类的外面访问数据成员。

在类的外面，不能仅仅使用数据成员的名称来访问它。例如，在程序中声明了一个名为 startingPoint 的 point 对象后，好像可以使用 x 和 y（在 point 类中的名称）来取得该点的坐标。但如果您在同一个程序中声明了 point 对象 startingPoint 和 endingPoint，将会出现什么情况呢？如果您使用 x，访问的将是哪个点的 x 坐标呢？

要访问数据成员，需要同时指定对象和数据成员的名称，并在它们之间加上成员运算符（句点）。因此，要访问 startingPoint 的坐标，可以使用下面的代码：

```
startingPoint.x
startingPoint.y
```

对于 endingPoint，则可以使用下面的代码：

```
endingPoint.x
endingPoint.y
```

现在，您具备了尝试编写一个程序所需的基本知识。程序清单 6.1 声明 point 类，并使用该类声明了两个对象：startingPoint 和 endingPoint。

程序清单 6.1 point.cs：声明包含数据成员的类

```
 1: // point.cs- A class with two data members
 2: //---------------------------------------------------
 3:
 4: class point
 5: {
 6:     public int x;
 7:     public int y;
 8: }
 9:
10: class pointApp
11: {
12:    public static void Main()
13:    {
14:       point starting = new point();
15:       point ending   = new point();
16:
17:       starting.x = 1;
18:       starting.y = 4;
19:       ending.x = 10;
20:       ending.y = 11;
21:
22:       System.Console.WriteLine("Point 1: ({0},{1})",
```

```
23:                             starting.x, starting.y);
24:       System.Console.WriteLine("Point 2: {0},{1}}",
25:                             ending.x, ending.y);
26:     }
27: }
```

该程序清单的输出如下：

```
Point 1: (1,4)
Point 2: (10,11)
```

分析：第4～8行声明了一个名为point的简单类，其结构与前面介绍的相同。第4行使用了关键字class，后面跟类名point。第5行和第8行是包围类体的花括号。在类体中，声明了两个int变量x和y。它们都被声明为公有的，以便能够在类的外面使用它们。

从第 10 行开始，是应用程序的主要部分，有意思的是这也是一个类！关于这一点，后面将做更详细的介绍。

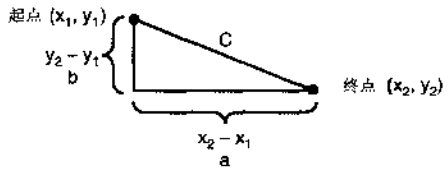
从第 12 行开始是一个例程，现在您应该对它很熟悉。第 14 和 15 行使用 point 类创建了两个对象，创建的格式与前面介绍的相同。第 17～20 行设置了每个 point 对象的数据成员的值。第 17 行将 starting 对象的数据成员 *x* 的值设置为 1。使用成员运算符将成员名和对象名分隔开来。第 18、19、20 行的格式与此相同。

第 22 行包含一个 WriteLine 例程，该例程在本书前面已经介绍过了。但这里很特别，因为打印的是存储在对象 starting 中的值。这些值被存储在 starting.x 和 starting.y 中，而不是 x 和 y 中。第 24 行打印 ending 对象的值。

6.4.2　使用数据成员

程序清单 6.1 演示了如何给数据成员赋值以及如何取得数据成员的值。如果要执行比赋值和显示更复杂的操作时，该如何办呢？

类的数据成员和其他任何类型的变量一样，您可以将其用于运算、控制语句或其他可以使用常规变量的任何地方。程序清单 6.2 扩展了 point 类的用途，计算了两点之间的距离。如果您忘记如何计算，请参考图 6.1，它说明了如何计算两点之间的距离。



$$c^2 = a^2 + b^2$$

或

$$c = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

图 6.1　计算两点之间的距离

程序清单 6.2　point2.cs：使用数据成员

```
1: // line.cs- Calculate the length of a line.
2: //---------------------------------------------------------
3:
```

```
 4: class point
 5: {
 6:     public int x;
 7:     public int y;
 8: }
 9:
10: class lineApp
11: {
12:    public static void Main()
13:    {
14:       point starting = new point();
15:       point ending   = new point();
16:       double line;
17:
18:       starting.x = 1;
19:       starting.y = 4;
20:       ending.x = 10;
21:       ending.y = 11;
22:
23:       line = System.Math.Sqrt( (ending.x - starting.x)*(ending.x -
starting.x) +
24:                               (ending.y - starting.y)*(ending.y - starting.y) );
25:
26:       System.Console.WriteLine("Point 1: ({0},{1})",
27:                               starting.x, starting.y);
28:       System.Console.WriteLine("Point 2: ({0},{1})",
29:                               ending.x, ending.y);
30:       System.Console.WriteLine("Length of line from Point 1 to Point 2: {0}",
31:                               line);
32:    }
33: }
```

该程序清单的输出如下所示：

```
Point 1: (1,4)
Point 2: (10,11)
Length of line from Point1 to Point 2: 11.4017542509914
```

分析：该程序清单与程序清单6.1极其类似。最大的不同在于增加了一个变量，并做了一些计算来确定线段的长度。第16行声明了一个新的名为line的double变量，用于存储两点之间的直线距离。

第 23 和 24 行实际上是一条语句，该语句看起来很复杂，其实并非如此。除了 System.Math.Sqrt 外，其他的内容您应该能够理解。Sqrt 是 System.Math 对象的一个例程，它计算平方根。将该公式与图 6.1 进行比较，您将发现这是正确的。最后得到的结果是线段的长度。需要注意的重要一点是，在计算中，数据成员的使用方式与其他变量相同，唯一的差别在于命名模式。

6.4.3 将类用作数据成员

前面介绍过，可以在一个类中嵌套另一个类。类也是一种数据类型，因此使用类声明的对象只不过是一种更复杂的变量而已，可以在能够使用其他变量的任何地方使用它。程序清单 6.3 包含一

个 line 类，这个类由两个 point 对象（starting 和 ending）组成。

程序清单 6.3　line2.cs：嵌套类

```
1:  // line2.cs- A class with two data members
2:  //-------------------------------------------------------------
3:
4:  class point
5:  {
6:     public int x;
7:     public int y;
8:  }
9:
10: class line
11: {
12:    public point starting = new point();
13:    public point ending = new point();
14: }
15:
16: class lineApp
17: {
18:   public static void Main()
19:   {
20:     line myLine = new line();
21:
22:     myLine.starting.x = 1;
23:     myLine.starting.y = 4;
24:     myLine.ending.x = 10;
25:     myLine.ending.y = 11;
26:
27:     System.Console.WriteLine("Point 1: {{0},{1}}",
28:                   myLine.starting.x, myLine.starting.y);
29:     System.Console.WriteLine("Point 2: {{0},{1}}",
30:                   myLine.ending.x, myLine.ending.y);
31:   }
32: }
```

该程序清单 的输出如下：

```
Point 1: {1,4}
Point 2: {10,11}
```

分析：该程序清单与前面的几个极其类似。第4～8行定义了point类，这与前面清单没有什么不同。但第10～14行定义了另一个类：line，它由两个point类变量（starting和ending）组成。当使用line类声明一个对象时，line类将创建两个point对象。

接下来，第 20 行创建了一个新的 line 对象 myLine，这里创建对象的格式与前面介绍的相同。

第 22～25 行访问 line 类的数据成员，并给它们赋值。咋一看，这要复杂些，但视觉是有欺骗性的。将其分解后，您将发现这些代码很直观。第 22 行将 1 赋给变量 myLine.starting.x，换句话说，是将 myLine 的 starting 成员的 x 成员设置为 1，也可以说是将 1 赋给了 myLine 对象的 starting 成员的

x 成员。这就像是一棵树。图 6.2 说明了 myLine 类的成员及其名称。

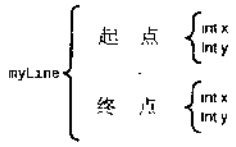


图 6.2 myLine 类的数据成员

6.4.4 类型嵌套

第 3 天介绍了可以使用的各种标准数据类型。正如您从程序清单 6.3 看到的，使用类创建的对象可以使用的地方与其他任何数据类型的变量相同。

就类本身而言，它并没有做任何工作，而只是一种描述。例如，在程序清单 6.3 中，第 4～8 行的 point 类只是一种描述，它没有声明任何东西，也没有使用内存。该描述定义了一种类型，这里是类，具体地说是 point 类。

可以将这种类型嵌套到其他类中。如果只在 line 中使用 point，则可以将 point 类的定义放在 line 类中。这样便可以在 line 类中使用 point 类。

嵌套 point 类的代码如下：

```
class line
{
     public class point
     {
     public int x;
     public int y;
     }
     public point starting = new point();
     public point ending = new point();
}
```

这里有些不同的地方是，point 类也必须定义为公有的。否则，将出错。这是有道理的，如果 point 本身都不是公有的，其组成部分如何能是公有的呢?

6.5 静态变量

有时候，您希望使用同一个类声明的对象可以共享一个值。例如，您可能声明许多起点都相同的 line 对象，如果其中的一个 line 对象修改了起点，则其他所有 line 对象的起点也将相应修改。

要在属于同一种类的对象之间共享数据值，可以使用限定符 static。程序清单 6.4 再次使用了 line 类，但所有的 line 类的起点都将相同。

程序清单 6.4 statline.cs：将限定符 static 用于数据成员

```
1: // statline.cs- A class with two data members
2: //-----------------------------------------------------
3:
4: class point
5: {
6:    public int x;
```

```
 7:    public int y;
 8: }
 9:
10: class line
11: {
12:    static public point origin= new point();
13:    public point ending = new point();
14: }
15:
16: class lineApp
17: {
18:   public static void Main()
19:   {
20:     line line1 = new line();
21:     line line2 = new line();
22:
23:     // set line origin
24:     line.origin.x = 1;
25:     line.origin.y = 2;
26:
27:
28:     // set line1's ending values
29:     line1.ending.x = 3;
30:     line1.ending.y = 4;
31:
32:     // set line2's ending values
33:     line2.ending.x = 7;
34:     line2.ending.y = 8;
35:
36:     // print the values...
37:     System.Console.WriteLine("Line 1 start: ({0},{1})",
38:                              line.origin.x, line.origin.y);
39:     System.Console.WriteLine("line 1 end:   ({0},{1})",
40:                              line1.ending.x, line1.ending.y);
41:     System.Console.WriteLine("Line 2 start: ({0},{1})",
42:                              line.origin.x, line.origin.y);
43:     System.Console.WriteLine("line 2 end:   ({0},{1})\n",
44:                              line2.ending.x, line2.ending.y);
45:
46:     // change value of line2's starting point
47:     line.origin.x = 939;
48:     line.origin.y = 747;
49:
50:     // and the values again...
51:
52:     System.Console.WriteLine("Line 1 start: ({0},{1})",
53:                              line.origin.x, line.origin.y);
54:     System.Console.WriteLine("line 1 end:   ({0},{1})",
55:                              line1.ending.x, line1.ending.y);
56:     System.Console.WriteLine("Line 2 start: ({0},{1})",
```

```
57:                              line.origin.x, line.origin.y);
58:       System.Console.WriteLine("line 2 end:   {0},{1}}",
59:                              line2.ending.x, line2.ending.y);
60:    }
61: }
```

该程序清单的输出如下：

```
Line 1 start: (1,2)
Line 1 end:  (3,4)
Line 2 start: (1,2)
Line 2 end:  (7,8)
Line 1 start: (939,747)
Line 1 end:  (3,4)
Line 2 start: (939,747)
Line 2 end:  (7,8)
```

警告：如果访问静态数据成员时使用对象名（如line1），将出错。访问静态数据成员时，必须使用类名。

分析：程序清单6.4与前面的几个差别不大，最大的差别在第12行，它除了将point声明为公有的外，还将其声明为静态的。static关键字使line类有天壤之别，这样使用line类声明的所有对象都共享同一个起点，而不是各有一个。

从第 18 行起是 Main 例程，第 20 和 21 行声明了两个名为 line1 和 line2 的 line 对象。第 28 ~ 29 行设置了 line 的终点，而第 33 ~ 34 行则设置了 line 的终点。第 24 和 25 行包含一些与以前不同的内容，它们设置了 line 类的起点坐标，而不是设置 line1 或 line2 的起点坐标。这很重要，如果试图设置 line1 或 line2 的起点坐标，则编译时将出错。换句话说，下面的代码是不正确的：

```
line1.origin.x = 1;
```

由于 origin 对象被声明为静态的，因此它将在所有的 line 对象之间共享。由于 line1 和 line2 都不拥有该对象，因此不能使用它们来直接设置该对象的值，而必须使用类名。记住，类中被声明为静态的变量属于这个类，而不是实例化的各个对象。

第 37 ~ 44 行打印 line1 和 line2 的起点和终点坐标，同样在打印起点坐标时，使用的是类名，而不是对象名。第 47 ~ 48 行修改起点的坐标，而后面的代码再次打印各线段的起点和终点坐标。

注意：静态数据成员的一种常见用途是用作计数器，每当某个对象执行某种操作时，便可以将所有对象的计数器加1。

6.6 应用程序类

您可能没有发现，在所有的应用程序中都需要使用应用程序类，对此我们还没有讨论过。程序清单 6.4 的第 16 行包含如下代码：

```
class lineApp
```

您将发现，在本书的每个应用程序中都包含一个与此类似的类。C#是一种面向对象的语言，这意味着所有的一切都是对象，应用程序也是。要创建对象，需要一个类来定义它。程序清单 6.4 的

应用程序类是 lineApp。当您执行该程序时，lineApp 类将被实例化，创建一个 lineApp 对象，它正是您的程序！

正如您在前面看到的，应用程序类声明数据成员。在程序清单 6.4 中，lineApp 类的数据成员是两个类：line1 和 line2。当然应用程序类还有其他功能，在明天的课程中您将知道，您的类中也可能包含这些功能。

6.7 属　性

前面指出过，面向对象程序的优点之一是能够控制数据的内部表示以及对数据的访问。在本章前面的范例中，所有的数据成员都被声明为公有的，因此可以在任何代码中访问这些数据成员。

在面向对象的程序中，您希望对哪些人可以访问（不可以访问）数据有更大的控制权。通常您不希望在类的外面可以直接访问数据成员，如果允许这样做，您将陷于无法改变数据成员的数据类型的境地。

C#提供了一种被称为属性（properties）的概念，让您能够在类中创建面向对象的字段（field）。属性使用关键字 get 和 set 来取得和设置变量的值。程序清单 6.5 演示了如何将 get 和 set 用于 point 类中。

程序清单 6.5　prop.cs：使用属性

```
1: // prop.cs- Using Properties
2: //---------------------------------------------------------
3:
4: class point
5: {
6:    int my_X; // my_X is private
7:    int my_Y; // my_Y is private
8:
9:    public int x
10:   {
11:     get
12:     {
13:       return my_X;
14:     }
15:     set
16:     {
17:       my_X = value;
18:     }
19:   }
20:   public int y
21:   {
22:     get
23:     {
24:       return my_Y;
25:     }
26:     set
27:     {
28:       my_Y = value;
```

```
29:       }
30:    }
31: }
32:
33: class MyApp
34: {
35:    public static void Main()
36:    {
37:       point starting = new point();
38:       point ending   = new point();
39:
40:       starting.x = 1;
41:       starting.y = 4;
42:       ending.x = 10;
43:       ending.y = 11;
44:
45:       System.Console.WriteLine("Point 1: ({0},{1})",
46:                                starting.x, starting.y);
47:       System.Console.WriteLine("Point 2: ({0},{1})",
48:                                ending.x, ending.y);
49:    }
50: }
```

该程序清单的输出如下：

```
Point 1: (1,4)
Point 2: (10,11)
```

分析：程序清单6.5为point类的x和y坐标分别创建了属性。Point类是在第4～31行定义的，这些代码行都是point的定义。第6～7行创建了两个数据成员my_x和my_y。因为它们没有被声明为公有的，因此在类的外面无法访问它们，所有被视为私有变量，第8天的课程将更详细地介绍如何使某些东西是私有的。

第 9～19 行和第 20～30 行的类似，只是前者使用的是变量 my_x，而后者使用的是变量 my_y。这些代码分别为 my_x 和 my_y 创建属性。

第 9 行看起来就像是声明另一个数据成员，事实上确实如此。这一行声明了一个名为 x 的 int 变量，注意该行的结尾没有分号，因此该成员变量声明还没有结束，它还包含第 10～19 行的代码块。该代码块中使用了两个命令，第 11 行是一条 get 语句，每当程序试图取得正在声明的数据成员（这里为 x）的值时，该语句便被调用。例如，如果您将 x 的值赋给其他变量，则首先需要取得 x 的值，然后将其放在该变量中。每当您设置变量 x 的值时，第 15 行的 set 语句都将被调用。例如将 x 的值设置为 10，将把 10 放在 my_x 中。

当程序要取得 x 的值时，第 11 行的 get 属性将被调用，这将执行 get 内的代码，即第 13 行。第 13 行返回 my_x 的值，而 my_x 是 point 类的私有变量。

当程序设置 x 的值时，第 15 行的 set 属性将被调用。这将执行 set 内的代码，即第 17 行。第 17 行将 value 的值赋给 point 类的私有变量 my_x。value 是 x 的值。例如，执行下面的语句后，value 的值将为 10：

```
x = 10;
```

该语句将 x 的值设置为 10，而 x 内的 set 属性将把这个值赋给 my_x。

从应用程序的主要部分（第 33～50 行）可以知道，x 的使用方式与以前相同。在使用 point 类方面，与以前没有任何差别。差别在于，可以修改 point 类，以不同的方式来存储 my_x 和 my_y，而不会对程序产生任何影响。

虽然第 9～30 行的代码很简单，但不一定非得如此。在 get 和 set 中，可以包含任何代码，也可以执行任何操作。您甚至不一定非得将值写入到另一个数据成员中。

应该
进入第6天的课程之前，务必要理解今天介绍的关于数据成员和类的知识
创建程序时，务必使用属性存取器（accessor）来访问类的数据成员

6.8 名称空间

开始学习类后，必须知道存在大量可以完成各种功能的类。.NET 框架提供了大量的基类供您使用。另外，您还可以获得第三方的类。

注意：第16天的课程将专门介绍如何使用许多重要的.NET基类。

随着您往下阅读本书，将了解很多重要的类。您实际上已经使用过几个基类。前面提到过，Console 是一个基类。您还知道，Console 有很多例程，其中的两个叫做 Write 和 WriteLine。例如，下面的代码将作者的姓名显示到控制台中：

```
System.Console.WriteLine("Bradley L. Jones");
```

您知道，“Bradley L. Jones”是一个字面值，而 WriteLine 是 Console 类的一个例程。您甚至还知道 Console 是一个使用类声明的对象。您唯一不了解的是 System。

由于类的数量非常大，因此以一定的方式对其进行组织至关重要。可以将类集中在一起，形成名称空间。名称空间是一组被命名的类。Console 类属于 Sytem 名称空间。

Sytem.Console.WriteLine 是一个全限定名称。使用全限定名称，可以直接指定代码所在的位置。C#提供一种使用类和方法的简捷方式，无需总是指定全限定名称。这是通过使用关键字 using 实现的。

关键字 using 让您能够在程序中包含名称空间，这样程序便知道到哪个名称空间中去搜索其中使用的类和例程。包含名称空间的格式如下：

```
using namespace_name
```

其中 namespace_name 是名称空间或嵌套名称空间的名称。例如，要包含名称空间 System，可以在程序清单的开始位置包含下面的代码：

```
using System;
```

包含上述代码后，在使用 System 名称空间中的类和例程时，便可以省略 System。程序清单 6.6 使用 using 语句来包含 System 名称空间。

程序清单 6.6 namesp.cs：使用 using 和名称空间

```
1: // namesp.cs- Namespaces and the using keyword
```

```
 2: //---------------------------------------------------
 3:
 4: using System;
 5:
 6: class name
 7: {
 8:    public string first;
 9:    public string last;
10: }
11:
12: class NameApp
13: {
14:   public static void Main()
15:   {
16:     // Create a name object
17:     name you = new name();
18:
19:     Console.Write("Enter your first name and press enter: ");
20:     you.first = Console.ReadLine();
21:     System.Console.Write("\n{0}, enter your last name and press enter: ",
22:                          you.first);
23:     you.last = System.Console.ReadLine();
24:
25:     Console.WriteLine("\nData has been entered.....");
26:     System.Console.WriteLine("You claim to be {0} {1}",
27:                              you.first, you.last);
28:   }
29: }
```

该程序清单的输出如下：

```
Enter your first name and press enter: Bradley

Bradley, enter your last name and press enter: Jones

Data has been entered.....
You claim to be Bradley Jones
```

注意：输出中的粗体文本是作者运行程序时输入的，您可以输入任何文本。建议您输入自己的姓名，而不是作者的。

分析：第4行是该程序的重点。关键字using将名称空间System包含进来，这样当您使用Console类中的函数时，便无需全限定其名称了，如第19、20和25行所示。使用using关键字后，仍可以使用全限定名称，如第21、23和26行所示。但没有必要这样做，因为名称空间已经被包含进来了。

该程序使用了 Console 类的另一个例程 ReadLine，通过运行这个程序可以知道，ReadLine 读取用户在按 Enter 键之前输入的信息，并返回用户输入的信息。在这个程序中，用户输入的信息被赋给 name 类的一个数据成员。

6.8.1　嵌套名称空间

多个名称空间可以存在一起，即作为一个名称存储。如果一个名称空间包含其他名称空间，则可以通过限定名称，或在 using 语句中使用子名称空间来包含它们。例如，System 名称空间包含其他几个名称空间，如 Drawing、Data 和 Windows.Forms，要使用这些名称空间中的类，可以使用限定名称，或使用 using 语句来包含它们。要使用 using 语句来包含 System 名称空间中的 Data 名称空间，可以使用下面的代码：

```
using System.Data;
```

6.9　总　结

今天和明天的课程是本书最重要的内容之一。类是面向对象编程语言的关键所在，因此也是 C#的关键所在。今天重温了封装、多态、继承和重用等概念。然后介绍了如何定义类的基本结构以及如何创建类的数据成员。学习如何使用存取器 set 和 get 创建属性时，您学会了最重要的、封装程序的方法之一。今天最后介绍了名称空间和 using 语句。

6.10　问与答

问：您是否会创建只包含数据成员的类？

答：通常不会。类和面向对象编程的价值在于能够将功能和数据封装到一个包中。今天只介绍了类的数据成员，明天将介绍如何给类添加功能。

问：是否应该将所有的数据成员都声明为公有的，以便其他人能够访问它们？

答：完全不是这样。虽然在今天的课程中，很多数据成员都被声明为公有的，但由于某些原因，有时候您不希望其他人能够访问您的数据。原因之一是以便能够修改数据的存储方式。

问：前面指出过，有很多已有的类，如何了解这些类？

答：微软公司提供了大量的、被称为.NET 基类的类，还提供了关于这些类的功能的文档。这些类是以名称空间的方式组织的，编写本书时，了解这些类的唯一途径是通过在线帮助。微软公司提供了关于这些基类的完全参考。第 19 天的课程将更详细地介绍这些基类。

6.11　作　业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

6.11.1　小测验

1. 面向对象程序的四个特征是什么？
2. 类中可以存储哪两种重要的东西？
3. 被声明为公有的数据成员和没有被声明为公有的数据成员之间的差别何在？
4. 给数据成员加上关键字 static 有何作用？
5. 程序清单 6.2 的应用程序类的名称是什么？
6. 哪些命令被用于实现属性？

7. value 有何用途?

8. Console 是类、数据成员、名称空间、例程还是数据类型?

9. System 是类、数据成员、名称空间、例程还是数据类型?

10. 使用哪个关键字来包含名称空间?

6.11.2 练习

1. 创建一个类来保存圆心和半径。

2. 在练习 1 创建的 Circle 类中添加一个属性。

3. 创建一个类，它存储一个 int 数据成员 MyNumber，并给该数据成员创建属性，当该数据成员被存储时，将其乘以 100；当其被读取时，将其除以 100。

4. 下面的程序有问题，请在编辑器中输入该程序，并编译它。哪一行导致错误消息?

```
1:// A bug buster program
2:// Is something wrong? Or not?
3://---------------------------------------
4: using System;
5: using System.Console;
6:
7: class name
8: {
9:    public string first;
10: }
11:
12: class NameApp
13: {
14:   public static void Main()
15:   {
16:     // Create a name object
17:     name you = new name();
18:
19:     Write("Enter your first name and press enter: ");
20:     you.first = ReadLine();
21:     Write("\nHello {0}!", you.first);
22:   }
23: }
```

5. 编写一个 die 类，它存储骰子的面数（sides）以及当前掷骰子得到的点数（value）。

6. 在程序中使用练习 5 的类声明两个骰子对象，并设置这些对象的 sides 数据成员，然后将骰子的点数设置为一个随机值。

第 7 天课程
类方法和成员函数

昨天介绍过，类有几个组成部分。最重要的是，类能够定义用于存储数据和例程的对象。昨天介绍了如何存储数据，今天将介绍如何在类中创建、存储和使用例程。例程让对象能够实现您所需的功能。虽然存储信息也很重要，但操作数据使程序具有生命力。今天将介绍以下内容：

- 创建自己的方法；
- 使用参数将信息传递给例程；
- 值和引用的概念；
- 调用方法的概念；
- 构造函数；
- 释放类。

7.1　方法初步

前几天介绍了如何存储和操纵数据以及如何控制程序流程。现在需要将其中的一些功能封装到可重用的例程中。另外，还需要将这些例程和类的数据成员关联起来。

在 C#中，例程被称为函数或方法，这两个术语之间并没有什么差别，可以交替使用。

注意： 大多数C++和C#开发人员将例程称为方法，而大多数C程序员将其称为函数。不管您管它们叫什么，指的都是同一回事。

新术语： 方法是一个带名称的、独立的、以可重用的方式放置的代码片段。方法无需应用程序其他部分的干预便能运行，如果创建的正确，则能够执行其名称指示的特定任务。

通过今天的课程，您将知道方法可以返回一个值。另外，许多方法还允许您传递信息给它们。

7.2　使用方法

在本书的前面，您已经使用过几个方法：Write、WriteLine 和 ReadLine。另外，在每一个程序中，您都使用了 Main 方法。程序清单 7.1 包含了前面介绍过的 Circle 类，不过这里将计算面积和周长的例程作为方法加入到了这个类中。

程序清单 7.1　circle.cs：一个包含成员方法的类

```
// circle.cs - A simple circle class with methods
//-----------------------------------------------

class Circle
{
   public int x;
   public int y;
   public double radius;

   public double area()
   {
      double theArea;
      theArea = 3.14159 * radius * radius;
      return theArea;
   }

   public double circumference()
   {
      double theCirc;
      theCirc = 2 * 3.14159 * radius;
      return theCirc;
   }
}

class CircleApp
{
  public static void Main()
  {
    Circle first  = new Circle();
    Circle second = new Circle();

    double area;
    double circ;
    first.x = 10;
    first.y = 14;
    first.radius = 3;

    second.x = 10;
    second.y = 11;
    second.radius = 4;

    System.Console.WriteLine("Circle 1: Center = ({0},{1})",
                             first.x, first.y);
    System.Console.WriteLine("        Radius = {0}", first.radius);
    System.Console.WriteLine("        Area   = {0}", first.area());
    System.Console.WriteLine("        Circum = {0}", first.circumfer- ence());

    area = second.area();
```

```
49:       circ = second.circumference();
50:
51:       System.Console.WriteLine("\nCircle 2: Center = ({0},{1})",
52:                                second.x, second.y);
53:       System.Console.WriteLine("        Radius = {0}", second.radius);
54:       System.Console.WriteLine("        Area   = {0}", area);
55:       System.Console.WriteLine("        Circum = {0}", circ);
56:    }
57: }
```

该程序清单的输出如下：

```
Circle 1: Center = (10,14)
       Radius = 3
       Area   = 28.27431
       Circum = 18.84954

Circle 2: Center = (10,11)
       Radius = 4
       Area   = 50.26544
       Circum = 25.13272
```

分析：该程序清单的大部分内容对您来说都是熟悉的，阅读完今天的课程后，那些不熟悉的部分便会变得熟悉了。

来看一下该程序清单，从第 4 行开始是 Circle 类的定义，第 6～8 行声明的三个数据成员与前一天的范例相同，它们是用于存储圆心坐标的 x 和 y 以及用于存储半径的变量 radius。数据成员声明之后的内容仍属于类定义。

第 10～15 行是第一个成员方法的定义，对其工作原理的详细解释见接下来的几节。但现在您知道该方法的名称为 area。第 12～14 行是该方法的代码，它们计算圆的面积并将其返回给调用程序。第 12～13 行您应该熟悉，关于第 14 行今天课程的后面将更详细地介绍。第 17～22 行是第二个方法，该方法名为 circumference，它计算圆的周长，并将其返回给调用程序。

从第 25 行开始是该程序清单的应用程序类。第 27 行包含 Main 方法，每个应用程序的入口都是 Main 方法。该例程创建两个 Circle 对象（第 29 和 30 行），然后给其数据成员赋值（第 34～40 行）。第 42 和 43 行打印第一个圆的数据成员。而第 45 和 46 行包含了以前使用过的 Console.WriteLine 方法，区别在于这里打印的是您传递的值。在第 45 行，您传递的是 first.area，这将调用第一个圆的成员方法 area，该方法是在第 10～15 行定义的。第 48 行调用第二个圆的 area 方法，并将结果赋给变量 area。

提示：在程序清单中，您知道area是一个成员方法，而不是数据成员，因为调用它时，后面跟有圆括号。后面将对此做更详细的介绍。

如果您还没有执行该程序清单，现在应该执行它，以看看发生的情况。接下来的几节将详细介绍如何定义自己的方法以及方法的工作方式。另外，还将介绍如何给方法传递值以及接收方法返回的值。

7.3 包含方法的程序的流程

前面介绍过，方法是一个独立的代码片段，被封装和命名，以便能够在程序中调用它。方法被调用时，程序流程将切换到方法，执行方法的代码，然后返回。图 7.1 说明了程序清单 7.1 的流程。方法被调用时，程序流程将切换到方法，执行方法的代码，然后返回到调用例程。另外，方法也可以调用其他的方法，流程与上面介绍的相同。

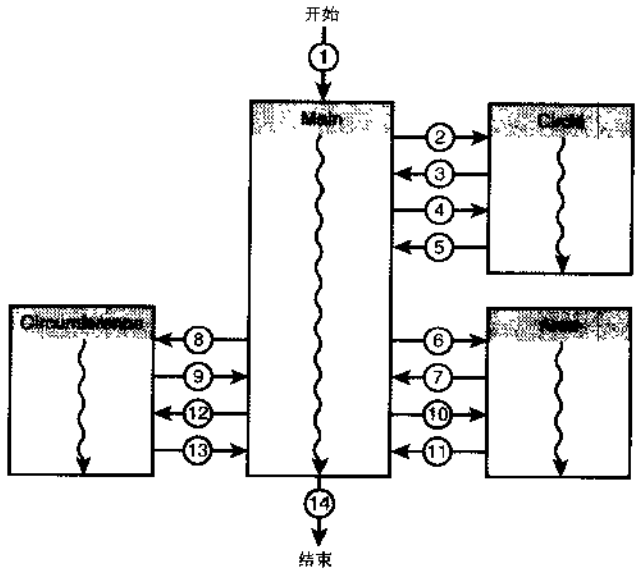


图 7.1 程序清单 7.1 的程序流程

7.4 方法的格式

了解方法的格式很重要。程序清单 7.1 以说明了调用方法的格式和步骤。方法的基本格式如下：

```
Method header
{
   Method body
}
```

7.4.1 方法头

方法头定义了关于方法的几项内容：

- 对方法的访问权限；
- 返回的数据类型；
- 传递给方法的值；
- 方法的名称。

从程序清单 7.1 的第 10 行可以知道，area 的方法头如下：

```
public double area( )
```

该方法被声明为公有的，即可以被类外面的程序访问，另外，该方法的返回数据类型为 double，即可以给调用程序返回一个 double 类型的值。该方法的名称为 area。最后，由于圆括号内为空，因此不用给 area 方法传递任何值，它使用同一个类的数据成员的值。稍后，将介绍如何给方法传递信息。

警告： *方法头不以分号结束，如果在方法头的结尾处加上分号，将出错。*

7.4.2　方法的返回数据类型

可以声明方法的返回类型，这是在方法头中指定的。方法的返回数据类型可以是任何合法的数据类型。

在方法体中，必须将一个返回数据类型的值返回给调用该方法的程序。要从方法返回一个值，可以使用关键字 return，后面跟一个数据类型为方法头中指定的值或变量。例如，在程序清单 7.1 中，area 方法的返回数据类型被声明为 double，而第 14 行使用关键字 return 返回了一个 double 类型的变量。area 方法将一个 double 值返回给调用程序。

如果方法不需要返回值，如何办呢？此时应使用哪种数据类型呢？如果方法不返回值，则使用关键字 void，该关键字指示方法不返回值。

7.4.3　给方法命名

给方法指定合适的名称很重要。在给方法命名方面，有多种理论。您需要确定哪种理论最适合您或您的单位。一种放之四海皆准的规则是，给方法取一个有意义的名称。如果方法计算并返回面积，则使用名称 area、GetArea、CalculateArea 和 CalcArea 等是有意义的。诸如 routine1 和 myRoutine 等的含义不明确。

对于给方法命名，一种流行的准则是，总是结合使用一个动词和一个名词。因为方法执行某种操作，因此总是可以组合使用一个动词和一个名词。按照这种准则，则 area 的含义不太明确，而 CalculateArea 和 CalcArea 等是更佳的选择。

7.4.4　方法体

方法体包含方法被调用时所执行的代码。这些代码以左花括号开始，以右花括号结束。花括号之间的代码可以包括本书前面介绍的任何编程元素。通常代码只修改其所属类的数据成员或传递给方法的数据。

如果方法头指示了返回数据类型，则方法返回的值的类型必须与之相同。您使用关键字 return 来返回值，关键字 return 的后面跟要返回的值。再来看一下程序清单 7.1 中的 area()方法，其方法体为第 11～15 行，它计算圆的面积，并将结果赋给 double 变量 theArea，然后在第 14 行使用 return 语句将这个值返回。

警告： *方法返回的变量的数据类型必须与方法头中指定的数据类型相同。*

7.4.4.1　使用方法

要使用方法，调用它即可。调用方法的方式与调用数据成员相同，即使用对象名和方法名，并在它们之间加上句点。调用方法和调用数据成员的区别在于，调用方法必须包含括号，还可能需要包含参数。在程序清单 7.1 中，使用下面的代码调用了第一个对象的 area 方法：

```
first.area( )
```

和变量一样，如果方法有返回值，则它是在方法被调用时返回的。例如，方法 area 将圆的面积

作为 double 值返回。在程序清单 7.1 的第 45 行，这个值被作为参数返回给另一个方法：Console.WriteLine。在第 48 行，第二个对象的 area 方法的返回值被赋给另一个变量：area。

7.4.4.2 在方法中使用数据成员

在程序清单 7.1 中，方法 area 使用了数据成员 radius，而未指定类名或对象名。该方法的代码如下：

```
public double area()
{
      double theArea;
      theArea = 3.14159 * radius * radius;
      return theArea;
}
```

以前，使用数据成员时，需要包含对象名，而这里使用 radius 时并没有包含对象名。例程为何可以省略对象名呢？仔细想想，则答案很简单。

当 area 方法被调用时，将使用特定的对象进行调用。当您使用 circle1 调用该方法时，circle1 中的所有常规数据成员和其他方法都将是可用的。之所以不需要使用对象名，是因为已经位于特定的对象的成员方法中。

另外，在类的成员方法中，还可以声明其他变量。这些变量只在方法运行时有效，它们被称为方法的局部变量。在 area 方法中，创建并使用了一个 double 变量 theArea。方法运行完毕后，存储在变量 theArea 中的值以及 theArea 变量本身都将不复存在。

程序清单 7.2 演示了局部变量的用法和程序流程。

程序清单 7.2 local.cs：使用局部变量和类变量

```
 1: // locals.cs - Local variables
 2: //---------------------------------------------------
 3:
 4: using System;
 5:
 6: class loco
 7: {
 8:    public int x;
 9:
10:    public void count_x()
11:    {
12:       int x;
13:
14:       Console.WriteLine("In count_x method. Printing X values...");
15:       for ( x = 0; x <= 10; x++)
16:       {
17:          Console.Write("{0} - ", x);
18:       }
19:       Console.WriteLine("\nAt the end of count_x method. x = {0}", x);
20:    }
21: }
22:
23: class CircleApp
```

```
24: {
25:    public static void Main()
26:    {
27:       loco Locals = new loco();
28:
29:       int x = 999;
30:       Locals.x = 555;
31:
32:       Console.WriteLine("\nIn Main(), x = {0}", x);
33:       Console.WriteLine("Locals.x = {0}", Locals.x);
34:       Console.WriteLine("Calling Method");
35:       Locals.count_x();
36:       Console.WriteLine("\nBack From Method");
37:       Console.WriteLine("Locals.x = {0}", Locals.x);
38:       Console.WriteLine("In Main(), x = {0}", x);
39:    }
40: }
```

该程序清单的输出如下：

```
In Main(), x = 999
Locals.x = 555
Calling Method
In count_x method. Printing X values...
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 -
At the end of count_x method. x = 11

Back From Method
Locals.x = 555
In Main(), x = 999
```

分析：该程序清单使用的各种名称的含义不明显，但说明了一些要点。该程序清单中声明了几个名为x的变量，这包括第8行声明的公有的int变量x（属于loco类的数据成员）、第12行声明的局部int变量x（属于方法count_x）和第29行声明的变量x（属于Main方法）。虽然三个变量的名称相同，但它们是不同的变量。

第一个 x 变量（属于 loco 类）最容易理解，它属于一个类。前面介绍过，要在 loco 类的外面使用该变量，必须指定对象名，如第 30 行所示，这里使用名为 Locals 的对象来将该对象的数据成员 x 的值设置为 555。在第 29 行，Main 例程的 x 变量的值被设置为 999。在第 32～33 行，这两个 x 变量包含各自的值，它们之间易于区分。

在第 35 行，Main 方法调用了第 27 行声明的 loco 对象 Locals 的 count_x 方法。进入该方法后，首先声明了一个名为 x 的变量（第 12 行），该变量将以前声明的所有 x 变量（包括类中声明的 x 变量）屏蔽掉。该方法的余下部分使用变量 x 进行循环并打印数字，循环结束后，将最后一次打印 x 的值，然后方法便执行完毕。

该方法执行完毕后，将返回到 Main 方法，再次打印变量 x 的值。Locals 的数据成员 x 以及 Main 内的局部变量 x 的值均未受影响，它们的值仍然分别是 555 和 999。这些 x 变量都是相互独立的。

如果要在 count_x 方法中使用类数据成员 x，该如何办呢？前面介绍过，在类的方法中使用数

据成员时，无需指定对象名。事实上，也无法使用对象名，因为对象名是不确定的。那么，当方法中包含局部变量 x 时，如何使用数据成员 x 呢？程序清单 7.3 对 7.2 稍做了修改。

程序清单 7.3 locals2.cs：在方法中使用数据成员

```
1: // locals.cs - Local variables
2: //---------------------------------------------------
3:
4: using System;
5:
6: class loco
7: {
8:    public int x;
9:
10:   public void count_x()
11:   {
12:     int x;
13:
14:     Console.WriteLine("In count_x method. Printing X values...");
15:     for ( x = 0; x <= 10; x++)
16:     {
17:       Console.Write("{0} - ", x);
18:     }
19:
20:     Console.WriteLine("\nDone looping.  x = {0}", x);
21:     Console.WriteLine("The data member x's value: {0}", this.x);
22:     Console.WriteLine("At the end of count_x method.");
23:   }
24: }
25:
26: class CircleApp
27: {
28:   public static void Main()
29:   {
30:     loco Locals = new loco();
31:
32:     int x = 999;
33:     Locals.x = 555;
34:
35:     Console.WriteLine("\nIn Main(), x = {0}", x);
36:     Console.WriteLine("Locals.x = {0}", Locals.x);
37:     Console.WriteLine("Calling Method");
38:     Locals.count_x();
39:     Console.WriteLine("\nBack From Method");
40:     Console.WriteLine("Locals.x = {0}", Locals.x);
41:     Console.WriteLine("In Main(), x = {0}", x);
42:   }
43: }
```

该程序清单的输出如下：

```
In Main(), x = 999
Locals.x = 555
Calling Method
In count_x method. Printing X values...
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 -
Done looping.  x = 11
The data member x's value: 555
At the end of count_x method.

Back From Method
Locals.x = 555
In Main(), x = 999
```

分析：该程序清单独特的地方是第21行，它打印一个名为this.x的值。关键字this指的是当前使用的对象。在这里，this指的是其count_x方法被调用的Locals对象。由于this指的是当前对象，因此x指的是该对象的数据成员x，而不是局部变量x。因此，要在方法中访问其所属类的数据成员，可以使用关键字this。

那么类的方法如何访问调用方法中的变量 x（在程序清单 7.3 的 Main 方法中声明的局部变量 x，如第 32 行所示）呢？除非将其作为参数传递给该方法，否则无法访问。

7.5　给方法传递值

您已经知道了如何访问方法、如何在方法中声明局部变量以及如何使用方法所属类的数据成员。如果要使用其他类或另一个方法的一个值或多个值，该如何办呢？例如，假设您需要一个接受两个数字，将其相乘并返回结果的方法。您已经知道如何返回单个值，但如何在方法中获得这两个数字呢？

方法被调用时，可以接受值。为此，必须在方法头中定义参数。包含参数的方法头的格式如下：

```
Modifiers ReturnType Name( Parameters)
```

参数是在括号内传递的，参数是可选的，如果没有参数，则括号内将为空，就像前面的范例那样。

传递参数的基本格式如下：

```
[Attribute] Type ArgumentName
```

其中 Type 是要传递的参数的数据类型，ArgumentName 是要传递的变量的名称。另外还可以指定属性，这将在本章后面介绍。属性是可选的。程序清单 7.4 中的程序包含一个简单的方法，它接受两个数字，将其相乘，然后将结果返回。

程序清单 7.4　Mult.cs：传递值

```
1: // mult.cs - Passing values
2: //-------------------------------------------------------
3:
4: using System;
5:
```

```
 6: class Multiply
 7: {
 8:    static long multi( long nbr1, long nbr2 )
 9:    {
10:       return (nbr1 * nbr2);
11:    }
12:
13:    public static void Main()
14:    {
15:       long x = 1234;
16:       long y = 5678;
17:       long a = 6789;
18:       long b = 9876;
19:
20:       long result;
21:
22:       result = multi( x, y);
23:       Console.WriteLine("x * y : {0} * {1} = {2}", x, y, result);
24:
25:       result = multi(a, b);
26:       Console.WriteLine("a * b : {0} * {1} = {2}", a, b, result);
27:
28:       result = multi( 555L, 1000L );
29:       Console.WriteLine("With Long values passed, the result was {0}", result);
30:    }
31: }
```

该程序清单的输出如下：

```
x * y : 1234 * 5678 = 7006652
a * b : 6789 * 9876 = 67048164
With Long values passed, the result was 555000
```

分析：该程序清单说明传递两个值的要点，同时说明了其他东西。首先来看一下第8～11行的方法定义，该方法名为Multi，接受两个参数。这两个参数都是long型的，分别名为nbr1和nbr2，它们是该方法内的局部变量。第10行将这两个值相乘，并将结果返回给调用者。在第8行的方法头中，方法multi的返回值被声明为long，因此它能够返回一个long值。

第 22、25 和 28 行使用不同的值分别调用了 multi 方法。从中可以知道，您可以传递变量（如第 22 和 25 行所示），也可以传递字面值（如第 28 行所示）。当 multi 方法被调用时，传递的值将被发送给该方法，并使用方法头中的变量名来引用它们。因此，对于第 22 行，x 和 y 分别被传递给 nbr1 和 nbr2；在第 25 行，a 和 b 分别被传递给 nbr1 和 nbr2；在第 28 行，555 和 1000 分别被传递给 nbr1 和 nbr2。然后，方法将使用这些传递来的值。

注意，传递给方法的参数数目必须与定义的相同。就 multi 方法而言，您必须传递两个值，否则将出错。

注意：对于使用像C++这样的语言编写过程序的读者，需要注意的是，在C#中，参数没有默认值。

7.5.1　静态方法

前面介绍过，static 限定符使数据成员与类而不是特定的对象关联在一起。程序清单 7.4 中使用了一个静态方法，该程序清单只包含一个类，其中不但包含 Main 方法，还包含 multi 方法。

也可以像前面那样，将 multi 方法放在另一个类中。如果这样做，则需要声明该类的一个对象，然后使用对象名调用 multi 方法。由于这里方法 multi 和 Main 位于同一个类中，因此将它声明为静态的，这样在 Multiply 类的任何地方都可以调用它，当然在 Main( )方法中也能使用它。

7.5.2　不同的参数访问属性

在前一个范例中，您给方法传递的是值。方法将获得这些值的一个拷贝，然后使用这些拷贝，当方法运行完毕后，这些拷贝将被丢弃。按值传递只是其中的一种传递方式，参数的访问属性有三类：

- 值；
- 引用；
- 输出（Out）。

7.5.2.1　以值的方式访问参数

正如前面指出的，以值的方式访问参数时，参数的一个拷贝将被传递给方法，然后方法使用这些拷贝，而原来的值不受影响。

7.5.2.2　以引用的方式访问参数

有时候，需要修改原来的变量中存储的值。在这种情况下，可以传递到变量的引用，而不是变量的值。引用是一个变量，它可以访问原来的变量。修改引用，将修改原来变量的值。

从技术上说，引用变量指向内存中的一个位置，其中存储了数据。请看图 7.2，变量的值存储在内存中，可以创建一个引用，它指向变量在内存中的位置。当引用被修改时，修改的是内存中的值，因此变量的值也将被修改。

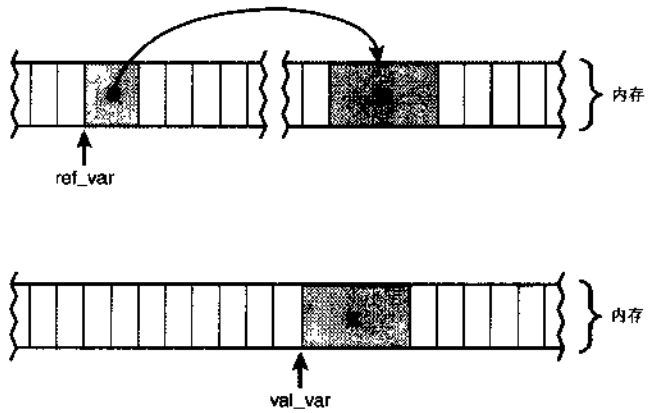


图 7.2　引用变量和值变量

由于引用变量可以指向不同的位置，因此它不像常规变量那样，总是对应于内存中的特定位置。当方法被调用时，如果该方法包含属性为引用的参数，则该参数将指向被传递给方法的相应变量。由于指向的是变量在内存中的位置，因此对参数变量的修改，也将导致原来的变量被相应修改。

声明参数时，默认情况下，其属性为其数据类型的属性。对于基本数据类型，为按值。要使基本数据类型（如 int）参数按引用传递，需要在方法头中的数据类型前面加上关键字 ref。程序清单

7.5 和 7.6 演示了如何使用关键字 ref 以及值参数和引用参数之间的区别。在程序清单 7.6 中，一个 double 参数是按引用传递的。从其输出可以知道这两个程序清单之间的区别。

注意： 正如第3天的课程介绍的，默认情况下，基本数据类型为值类型。这意味着，当您创建一个变量时，系统将从内存中分配特定的单元，供它存储自己的值。诸如类等数据类型默认为引用类型，这意味着类名包含的是类数据的存储地址，而不是数据本身。

程序清单 7.5 refnot.cs：按值传递参数

```
// refnot.cs - without reference variables (by value)
//-----------------------------------------------

using System;

class nbr
{
   public double square( double x )
   {
      x = x * x;
      return x;
   }
}

class TestApp
{
  public static void Main()
  {
    nbr doit = new nbr();

    double nbr1 = 3;
    double retVal = 0;

    Console.WriteLine("Before square -> nbr1 = {0}, retVal = {1}",
                      nbr1, retVal);

    retVal = doit.square( nbr1 );

    Console.WriteLine("After square -> nbr1 = {0}, retVal = {1}",
                      nbr1, retVal);
  }
}
```

该程序清单的输出如下：

```
Before square -> nbr1 = 3, retVal = 0
After square -> nbr1 = 3, retVal = 9
```

程序清单 7.6 refers.cs：按引用传递参数

```
// refers.cs - Reference variables
//-------------------------------------------------
```

```
 3:
 4: using System;
 5:
 6: class nbr
 7: {
 8:    public double square( ref double x )
 9:    {
10:       x = x * x;
11:       return x;
12:    }
13: }
14:
15: class TestApp
16: {
17:   public static void Main()
18:   {
19:      nbr doit = new nbr();
20:
21:      double nbr1 = 3;
22:      double retVal = 0;
23:
24:      Console.WriteLine("Before square -> nbr1 = {0}, retVal = {1}",
25:                        nbr1, retVal);
26:
27:      retVal = doit.square( ref nbr1 );
28:
29:      Console.WriteLine("After square -> nbr1 = {0}, retVal = {1}",
30:                        nbr1, retVal);
31:   }
32: }
```

该程序清单的输出如下：

```
Before square -> nbr1 = 3, retVal = 0
After square -> nbr1 = 9, retVal = 9
```

分析：这两个程序清单的输出说明了发生的情况，程序清单7.5使用的不是引用，因此传递给square方法的变量的值没变，在方法调用前后，该变量的值都是3。在程序清单7.6中，传递的是一个引用，因此传递给square的变量的值被修改，如该程序清单的输出所示。

在程序清单 7.6 中，在方法头中声明参数时，加上了关键字 ref。正常情况下，double 是一种值数据类型，因此，要将其作为引用传递给方法，必须在被传入的变量前面加上关键字 ref，如第 27 行所示。

7.5.2.3　以输出方式访问参数

通过指定返回类型，可以从方法返回一个值，但有时候，需要返回多个值　虽然可以使用引用变量来完成这样的工作，但 C#提供了一种特殊的属性类型，专门用于从方法返回数据。

可以在方法头中加入专门用于返回值的参数。这种参数的声明方式与其他参数相同，只是需要将其属性设置为关键字 out。该关键字指示从方法返回一个值，但不需要传入。调用包含输出参数的

方法时，一定要包含用于存储返回值的变量。程序清单 7.7 演示了如何使用 out 属性。

程序清单 7.7 Outter.cs：使用 out 属性

```
// outter.cs - Using output variables
//---------------------------------------------------------

using System;

class nbr
{
   public void math_routines( double x,
                              out double half,
                              out double squared,
                              out double cubed )
   {
     half = x / 2;
     squared = x * x;
     cubed = x * x * x;
   }
}

class TestApp
{
  public static void Main()
  {
    nbr doit = new nbr();

    double nbr = 600;
    double Half_nbr = 0;
    double Squared_nbr = 0;
    double Cubed_nbr = 0;

    Console.WriteLine("Before method -> nbr = {0}", nbr);
    Console.WriteLine("          Half_nbr = {0}", Half_nbr);
    Console.WriteLine("       Squared_nbr = {0}", Squared_nbr);
    Console.WriteLine("        Cubed_nbr = {0}\n", Cubed_nbr);

    doit.math_routines( nbr, out Half_nbr, out Squared_nbr, out Cubed_nbr );

    Console.WriteLine("After method -> nbr = {0}", nbr);
    Console.WriteLine("         Half_nbr = {0}", Half_ubr);
    Console.WriteLine("      Squared_nbr = {0}", Squared_nbr);
    Console.WriteLine("       Cubed_nbr = {0}", Cubed_nbr);
  }
}
```

该程序清单的输出如下：

```
Before method -> nbr = 600
        Half_nbr = 0
```

```
        Squared_nbr = 0
          Cubed_nbr = 0

After method -> nbr = 600
          Half_nbr = 300
       Squared_nbr = 360000
         Cubed_nbr = 216000000
```

分析：该程序清单中需要说明的有两部分。首先是第8～11行的方法头。还记得吗，可以使用空白来提供代码的可读性。该方法被格式化成每个参数占据一行。第一个参数是一个常规的double变量，其他三个参数的属性被声明为out。这意味着，这三个参数不是将值传入到方法内，而是从方法中传出值。

需要说明的第 2 部分代码是第 35 行，它调用方法 math_routines。调用方法时，传递的参数的属性也必须指定为 out，如果省略关键字 out，将出错。

就整体而言，该程序清单还是相对比较简单。nbr 类中的 math_routines 使用一个数字执行三种运算。第 30～33 行打印用来存储数学计算结果的变量的值。第 37～40 行打印这些变量在 math_routines 方法中被填充后的值。

如果不给输出参数设置值，将出错。必须给方法的所有输出参数赋值。例如，请将程序清单的第 14 行注释掉：

```
14:         squared = x * x;
```

这样输出变量 squared 将不被赋值。重新编译该程序清单时，将出现如下所示的错误：

```
outterz.cs(8,17): error CS0177: The out parameter 'squared' must be assign to before control
leaves the current method
```

从上面可以知道，如果定义了一个输出参数，则必须给它赋值。

注意：将变量用作输出变量时，在调用方法前，可以不初始化。例如，可以删除第26～28行的初始化部分，变成如下所示：

```
26:         double Half_nbr;
27:         double Squared_nbr;
28:         double Cubed_nbr;
```

当然，在这种情况下，还需要删除第31～33行。重新编译该程序清单，第35行的方法调用将不会出现任何问题，虽然这三个变量没有被初始化。

应该	不应该
应给类和方法指定易懂的、描述性的名称	不要将值变量和引用变量混淆，按值传递变量将创建变量的一个拷贝，按引用传递变量将可以操纵原来变量的值

7.6　类方法的类型

前面介绍了使用方法的基本知识，您还需了解一些特殊的方法：

- 属性存取器方法；

- 构造函数；
- 析构函数。

7.6.1 属性存取器方法

您在昨天实际上使用过属性存取器方法：set 和 get，这些方法让您能够使数据成员为私有的。

7.6.2 构造函数

有时候，在对象被创建时，您想做一些设置。有一种特殊的方法专门用于完成对象初始设置或构造，这种方法叫做构造函数。实际上存在两种构造函数：实例构造函数和静态构造函数，前者在每个实例或对象创建时被调用，后者在类的第一个对象被创建之前被调用。

7.6.2.1 实例构造函数

实例构造函数是一个方法，每当实例化一个对象时，该方法都将被调用。这种构造函数可以包含常规函数能包含的任何代码。构造函数通常用于完成对象的初始化设置，可以包含诸如初始化变量等功能。

构造函数的格式如下：

```
modifiers classname( )
{
   // Constructor body
}
```

构造函数的名称与其所属的类相同，其中限定符与用于其他方法的限定符相同。通常只使用 public，不指定返回数据类型。

需要注意的是，每个类都有一个默认的构造函数，即使您没有创建任何构造函数。通过创建构造函数，可以控制某些设置。

每当您创建一个对象时，相应类的构造函数都将被调用。程序清单 7.8 演示了如何使用构造函数。

程序清单 7.8 Constr.cs：使用构造函数

```
 1: // constr.cs - constructors
 2: //---------------------------------------------------------
 3:
 4: using System;
 5:
 6: public class myClass
 7: {
 8:     static public int sctr = 0;
 9:     public int ctr = 0;
10:
11:     public void routine()
12:     {
13:       Console.WriteLine("In the routine - ctr = {0} / sctr = {1}\n",
14:                         ctr, sctr );
15:     }
16:
17:     public myClass()
18:     {
```

```
19:        ctr++;
20:        sctr++;
21:        Console.WriteLine("In Constructor- ctr = {0} / sctr = {1}\n",
22:                          ctr, sctr );
23:     }
24: }
25:
26: class TestApp
27: {
28:    public static void Main()
29:    {
30:       Console.WriteLine("Start of Main method...");
31:
32:       Console.WriteLine("Creating first object...");
33:       myClass first = new myClass();
34:       Console.WriteLine("Creating second object...");
35:       myClass second = new myClass();
36:
37:       Console.WriteLine("Calling first routine...");
38:       first.routine();
39:
40:       Console.WriteLine("Creating third object...");
41:       myClass third = new myClass();
42:       Console.WriteLine("Calling third routine...");
43:       third.routine();
44:
45:       Console.WriteLine("Calling second routine...");
46:       second.routine();
47:
48:       Console.WriteLine("End of Main method");
49:    }
50: }
```

该程序清单的输出如下所示：

```
Start of Main method...
Creating first object...
In Constructor- ctr = 1 / sctr = 1

Creating second object...
In Constructor- ctr = 1 / sctr = 2

Calling first routine...
In the routine - ctr = 1 / sctr = 2

Creating third object...
In Constructor- ctr = 1 / sctr = 3

Calling third routine...
In the routine - ctr = 1 / sctr = 3
```

```
Calling second routine...
In the routine - ctr = 1 / sctr = 3

End of Main method
```

分析：该程序清单演示一个非常简单的构造函数，如第17~23行所示，它还再次说明了静态数据成员和常规数据成员之间的差别。第6~24行定义了一个名为myClass的类，这个类包含两个被用作计数器的数据成员，第一个数据成员被声明为静态的，名称为sctr；第二个不是静态的，名为ctr（不带s）。

注意：类将创建一个在所有对象之间共享的静态数据成员拷贝，而对于常规数据成员，每个对象都有自己的一个拷贝。

myClass 类包含两个方法。第一个叫 routine，它打印一行文本，其中包含两个计数器的当前值；第二个方法的名称与类名相同，即为 myClass，通过这一点，您知道它是一个构造函数。每当对象被创建时，该方法都将被调用。该方法完成了几项工作。首先将每个计数器的值加 1。ctr 是首次被加 1，因为它是特定对象的 ctr 拷贝。sctr 则不同，因为它是该类的所有对象共享的。因此，每创建该类的一个对象，sctr 的值都将被加 1。最后，第 21 行打印一条消息，显示 ctr 和 sctr 的值。

从第 26 行开始是该程序的应用程序类，它打印消息并实例化 myClass 对象，仅此而已。查看输出中的消息，您将发现，这与程序清单是相符的。需要注意的有趣的一点是，在第 46 行调用第二个对象的 routine 方法时，sctr 的值为 3，而不是 2。这是因为 sctr 是在所有的对象之间共享的，当您打印该消息时，您已经调用了构造函数三次。

最后需要注意的是，不要在程序清单中调用构造函数，请看第 33 行：

```
33:       myClass first = new myClass();
```

这行代码创建一个对象。虽然构造函数是自动调用的，但请注意这行中的 myClass 调用！

注意：明天的课程将介绍如何给构造函数传递参数。

7.6.2.2 静态构造函数

与数据成员和方法一样，也可以创建静态构造函数。使用限定符 static 声明的构造函数将在第一个对象被创建之前被调用，它只被调用一次，以后便不再被使用。程序清单 7.9 对程序清单 7.8 做了一些修改。在该程序清单中，在 test 类中添加了一个静态构造函数。

注意，这个构造函数同其他构造函数的名字相同。因为静态构造函数的名字包括 Static，编译器会将其与普通的构造函数区分开。

程序清单 7.9 statcon.cs：使用静态构造函数

```
1: // statcon.cs - static constructors
2: //-------------------------------------------
3:
4: using System;
5:
6: public class test
7: {
8:     static public int sctr;
9:     public int ctr;
```

```
10:
11:     public void routine()
12:     {
13:       Console.WriteLine("In the routine - ctr = {0} / sctr = {1}\n",
14:                         ctr, sctr );
15:     }
16:
17:     static test()
18:     {
19:       sctr = 100;
20:       Console.WriteLine("In Static Constructor - sctr = {0}\n", sctr );
21:     }
22:
23:     public test()
24:     {
25:       ctr++;
26:       sctr++;
27:       Console.WriteLine("In Constructor- ctr = {0} / sctr = {1}\n",
28:                         ctr, sctr );
29:     }
30: }
31:
32: class TestApp
33: {
34:   public static void Main()
35:   {
36:     Console.WriteLine("Start of Main method...");
37:
38:     Console.WriteLine("Creating first object...");
39:     test first = new test();
40:     Console.WriteLine("Creating second object...");
41:     test second = new test();
42:
43:     Console.WriteLine("Calling first routine...");
44:     first.routine();
45:
46:     Console.WriteLine("Creating third object...");
47:     test third = new test();
48:     Console.WriteLine("Calling third routine...");
49:     third.routine();
50:
51:     Console.WriteLine("Calling second routine...");
52:     second.routine();
53:
54:     Console.WriteLine("End of Main method");
55:   }
56: }
```

该程序清单的输出如下：

```
Start of Main method...
Creating first object...
In Static Constructor - sctr = 100

In Constructor- ctr = 1 / sctr = 101

Creating second object...
In Constructor- ctr = 1 / sctr = 102

Calling first routine...
In the routine - ctr = 1 / sctr = 102

Creating third object...
In Constructor- ctr = 1 / sctr = 103

Calling third routine...
In the routine - ctr = 1 / sctr = 103

Calling second routine...
In the routine - ctr = 1 / sctr = 103

End of Main method
```

分析：该程序清单的输出和程序清单7.8没有什么本质的不同。输出中的第三行来自静态构造函数。该构造函数（第17～21行）将静态数据成员sctr初始化为100，然后打印一条消息。程序其他部分的运行方式与程序清单7.8完全相同。输出与程序清单7.8稍有不同的原因在于，变量sctr的初始值为100，而不是0.

7.6.3 析构函数

可以在对象被释放时执行某些操作，这是在析构函数中实现的。

当程序不再使用对象后的某一时刻，析构函数将自动执行。所谓“某一时刻”是不是有些不明确？这是故意的，析构操作可以在对象不再被使用到程序结束之前的任何时候进行。事实上，程序结束时也可能不调用析构函数，这意味着析构函数永远不会被调用。至于何时执行析构函数，您并没有真正的控制权，因此析构函数的价值有限。

警告：在诸如C++这样的语言中，可以调用析构函数，程序员能够控制何时执行析构函数。而在C#中，不是这样。

从技术的角度

析构函数通常是由 C#运行阶段环境在对象不再被使用时调用的。C#运行阶段环境通常是在检查是否有内存可供释放（被称为无用单元收集）前调用析构函数。如果在对象不再使用之后到程序结束之前，C#运行阶段环境没有执行这种内存检查，则析构函数将永远不会被调用。可以强行进行无用单元收集，但这还不如尽量少用析构函数。

7.6.3.1 使用析构函数

在C#中，析构函数是使用后跟类名和空括号的代字号（~）来定义的。例如，类 xyz 的析构函

数为：

```
~xyz( )
{
   // Destructor body
}
```

析构函数不使用限定符或其他关键字。程序清单 7.10 对 7.8 进行了简化，并添加了一个析构函数。

程序清单 7.10：destr.cs：使用析构函数

```
 1: // destr.cs - constructors
 2: //---------------------------------------
 3:
 4: using System;
 5:
 6: public class test
 7: {
 8:    static public int sctr = 0;
 9:    public int ctr = 0;
10:
11:    public void routine()
12:    {
13:       Console.WriteLine("In the routine - ctr = {0} / sctr = {1}",
14:                         ctr, sctr);
15:    }
16:
17:    public test()
18:    {
19:       ctr++;
20:       sctr++;
21:       Console.WriteLine("In Constructor");
22:    }
23:
24:    ~test()
25:    {
26:       Console.WriteLine("In Destructor");
27:    }
28: }
29:
30: class TestApp
31: {
32:   public static void Main()
33:   {
34:      Console.WriteLine("Start of Main method");
35:
36:      test first = new test();
37:      test second = new test();
38:
39:      first.routine();
```

```
40:
41:         test third = new test();
42:         third.routine();
43:
44:         second.routine();        // calling second routine last
45:
46:         Console.WriteLine("End of Main method");
47:     }
48: }
```

该程序清单的输出如下：

```
Start of Main method
In Constructor
In Constructor
In the routine - ctr = 1 / sctr = 2
In Constructor
In the routine - ctr = 1 / sctr = 3
In the routine - ctr = 1 / sctr = 3
End of Main method
In Destructor
In Destructor
In Destructor
```

从输出可以知道，在每个对象被释放后，析构函数被调用。在这里，是在 Main()方法结束时；但也有可能析构函数永远不被调用。

注意： 需要重申的是，析构函数不一定被调用。当您执行程序清单7.10时，便可能发现它们没有被执行。

7.6.3.2 析构函数和扫尾工作

由于内部发生的一些事情，析构函数常被称为结束器（finalizer）。析构函数与一个叫作 Finalize 的方法有关。为完成扫尾工作（finalization），编译器将把构造函数转换为正确的代码。

7.7 总 结

今天的课程介绍的主题不多，但这些主题对于使用 C#和面向对象语言编程举足轻重。昨天介绍了如何在类中添加数据成员，而今天介绍了如何以方法的方式给类添加功能。您知道了方法、函数和例程虽然是不同的术语，但指的是同一回事。

介绍方法的基本知识后，重温了“按值”和“按引用”之间的差别，介绍了如何以参数（argument）的方式将信息传递给方法头中指定的参量（parameter）。通过使用诸如 ref 和 out 等关键字，可以改变方法处理被传递的数据的方式。

最后，介绍了一些特殊的方法，包括构造函数和析构函数。

本周复习的后面是明天的课程。在第 8 天的课程中，将扩充您今天学到关于方法的知识，将介绍重载、代表以及方法的许多其他的高级特性。

7.8 问与答

问：实参（argument）和形参（parameter）之间的区别何在？

答：形参定义了应将什么传递给方法，它位于方法头的方法定义中。实参是传递给方法的值，传递给方法的实参必须与方法定义中指定的形参匹配。

问：可以在类外创建方法吗？

答：虽然在其他语言中，可以在类的外面创建方法，但在 C#中不可以。C#是面向对象的，因此所有的代码都必须位于类框架之内。

问：在 C#中，方法和类的运行方式与在诸如 C++和 Java 等语言中相同吗？

答：在很大程度上是类似的，但随语言而异。详细介绍这种差别超出了本书的范围。其中的差别之一是，C#不允许方法的参数有默认值。在诸如 C++等语言中，如果调用方法时没有给参数提供值，则该参数可以有默认的特定值。而 C#则不是这样。当然还有其他差别。

问：如果不打算指望析构函数，如何编写扫尾代码？

答：建议创建自己的代码来完成扫尾工作，并在您知道不会再使用某个对象时，显式地调用这些代码。例如，如果有一个创建文件对象的类，则在使用完这种对象后，应该关闭文件。由于析构函数可能不会被调用，或者很长时间后才会被调用，因此应该创建自己的关闭文件的方法。您确实不希望使用完文件后，还让它处于打开状态。

7.9 作　业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

7.9.1 小测验

1. 方法的两个重要组成部分是什么？
2. 函数和方法之间有何区别？
3. 方法可以返回多少个值？
4. 使用哪个关键字从方法返回一个值？
5. 方法可以返回哪些数据类型？
6. 访问类的成员方法使用什么样的格式？ 例如，如果使用 myClass 类实例化了一个名为 myObject 的对象，而该类包含一个名为 myMethod 的方法，下面哪种访问该方法的方式是正确的？

a. myClass.myMethod

b. myObject.myMethod

c. myMethod

d. myClass.myObject.myMethod

7. 按引用传递变量和按值传递变量之间的区别何在？
8. 构造函数在何时被调用？
9. 没有参数的析构函数的语法是什么样的？
10. 析构函数在何时被调用？

7.9.2 练习

1. 编写公有方法 xyz 的方法头，该方法接受两个参数，不返回任何值。

2. 为方法 myMethod 编写方法头。该方法接受三个参数。第一个名为 myVal，其数据类型为 double，并按值传递；第二个是一个输出变量，名为 myOutput；第三个是按引用传递的，数据类型为 int，名为 myReference。另外该方法是公有的，其返回类型为 byte。

3. 使用程序清单 7.1 中的 Circle 类，添加一个构造函数，将默认的圆心坐标设置为(5,5)，半径为 1。在程序中使用这个类，该程序打印缺省的值和您设置的值。不要在 Main( )方法中打印关于圆的信息，而是创建一个打印圆信息的方法。

4. 下面的代码片段有问题，哪一行将导致错误消息？

```
public void myMethod()
{
     System.Console.WriteLine("I'm a little teapot short and stout");
     System.Console.WriteLine("Down came the rain and washed the spider out");

     return 0;
}
```

5. 使用几天前介绍的骰子类，创建一个新的程序。该程序包含一个骰子类，后者有三个数据成员，即骰子的面数、骰子的点数以及包含随机数类（在前面的范例中，被定义为 rnd）的静态数据成员。为这个类声明一个名为 roll( )的成员方法，它以随机的方式返回下一次掷骰子得到的点数。

第 1 周复习

恭喜您！您完成了学习 C#的第 1 周课程。这一周为编写 C#应用程序打下了基础。您学习了如何存储基础数据、控制程序流程、重复执行代码片段、创建包含数据和方法的类以及许多其他的知识。

本周介绍的大部分清单和程序都是重点介绍某个概念。下面的代码将本周学习的各种概念综合在一起，正如您看到的，程序清单将更长一些。

执行该程序清单时，屏幕上将出现一个菜单，供您选择其中的选项。系统将根据您的选择来创建类并执行一些代码。

注意：该程序清单没有包含任何您还未学习过的东西。在接下来的两周中，您将学习一些对该程序清单进行改进的办法。这包括以更佳的方式实现某些功能、其他一些取得并转换来自用户的数据的方式，等等。

程序 WR01.cs

请输入、编译并执行程序清单 WR01.cs。该清单中包含 XML 注释，这意味着您可以通过包含第 2 天介绍的编译器开关/doc 来生成 XML 文档。

注意：虽然作者认为，学习的最佳方式是输入程序清单，并犯一些错误，但可以从Sam出版社的网站下载本书中各个程序清单的源代码。

程序清单 WR1.1　WR01.cs：第 1 周复习

```
1:  // File:  wr01.cs
2:  // Desc:  Week One In Review
3:  //        This program presents a menu and lets the user select a c
4:  //        a choice. Based on this choice, the program then executes
5:  //        a set of code that either manipulates a shape or exits the
6:  //        program
7:  //--------------------------------------------------------
8:
9:  using System;
10:
11: class WR01App
```

```
12: {
13:    /// <summary>
14:    /// Main() routine that starts the application
15:    /// </summary>
16:    public static void Main()
17:    {
18:      int menuChoice = 99;
19:
20:      do
21:      {
22:        menuChoice = GetMenuChoice();
23:
24:        switch( menuChoice )
25:        {
26:          case 0:  break;
27:          case 1:  WorkWithLine();
28:                   break;
29:          case 2:  WorkWithCircle();
30:                   break;
31:          case 3:  WorkWithSquare();
32:                   break;
33:          case 4:  WorkWithTriangle();
34:                   break;
35:          default: Console.WriteLine("\n\nError... Invalid menu option.");
36:                   break;
37:        }
38:
39:        if ( menuChoice !=0 )
40:        {
41:          Console.Write("\nPress <ENTER> to continue...");
42:          Console.ReadLine();
43:        }
44:
45:      } while ( menuChoice != 0 );
46:    }
47:
48:    /// <summary>
49:    /// Displays a menu of choices.
50:    /// </summary>
51:    static void DisplayMenu()
52:    {
53:      Console.WriteLine("\n          Menu");
54:      Console.WriteLine("=========================\n");
55:      Console.WriteLine(" A - Working with Lines");
56:      Console.WriteLine(" B - Working with Circles");
57:      Console.WriteLine(" C - Working with Squares");
58:      Console.WriteLine(" D - Working with Triangles");
59:      Console.WriteLine(" Q - Quit\n");
60:      Console.WriteLine("=========================\n");
61:    }
```

```
62:
63:    /// <summary>
64:    /// Gets a choice from the user and verifies that it is valid.
65:    /// Returns a numeric value to indicate which selection was made.
66:    /// </summary>
67:    static int GetMenuChoice()
68:    {
69:      int option = 0;
70:      bool cont = true;
71:      string buf;
72:
73:      while( cont == true )
74:      {
75:        DisplayMenu();
76:        Console.Write(" Enter Choice: ");
77:        buf = Console.ReadLine();
78:
79:        switch( buf )
80:        {
81:          case "a":
82:          case "A": option = 1;
83:                    cont = false;
84:                    break;
85:          case "b":
86:          case "B": option = 2;
87:                    cont = false;
88:                    break;
89:          case "c":
90:          case "C": option = 3;
91:                    cont = false;
92:                    break;
93:          case "d":
94:          case "D": option = 4;
95:                    cont = false;
96:                    break;
97:          case "q":
98:          case "Q": option = 0;
99:                    cont = false;
100:                   break;
101:
102:         default:  Console.WriteLine("\n\n--> {0} is not valid <-- \n\n", buf);
103:                   break;
104:       }
105:     }
106:     return option;
107:   }
108:
109:   /// <summary>
110:   /// Method to perform code for Working with Line.
111:   /// </summary>
```

```
112:    static void WorkWithLine()
113:    {
114:       line myLine = new line();
115:
116:       myLine.start.x = 0;
117:       myLine.start.y = 0,
118:       myLine.end.x = 3;
119:       myLine.end.y = 3;
120:
121:       myLine.DisplayInfo();
122:    }
123:
124:    /// <summary>
125:    /// Method to perform code for Working with Circles.
126:    /// </summary>
127:    static void WorkWithCircle()
128:    {
129:       circle myCircle = new circle();
130:
131:       myCircle.center.x = 1;
132:       myCircle.center.y = 1;
133:       myCircle.radius = 10;
134:
135:       myCircle.DisplayInfo();
136:    }
137:
138:    /// <summary>
139:    /// Method to perform code for Working with Squares.
140:    /// </summary>
141:    static void WorkWithSquare()
142:    {
143:       square mySquare = new square();
144:
145:       mySquare.width.start.x = 1;
146:       mySquare.width.end.x = 10;
147:       mySquare.width.start.y = 0;
148:       mySquare.width.end.y = 0;
149:       mySquare.height.start.y = 2;
150:       mySquare.height.end.y = 8;
151:       mySquare.height.start.x = 0;
152:       mySquare.height.end.x = 0;
153:
154:       mySquare.DisplayInfo();
155:     }
156:
157:    /// <summary>
158:    /// Method to perform code for Working with Triangles.
159:    /// </summary>
160:    static void WorkWithTriangle()
161:    {
```

```
162:       Console.WriteLine("\n\nDo Triangle Stuff\n\n");
163:       // This section left for you to do
164:    }
165: }
166:
167: //-----------------------------------------------------------
168: /// <summary>
169: /// This is a point class. It is for storing and
170: /// working with an (x,y) value.
171: /// </summary>
172: class point
173: {
174:    private int point_x;
175:    private int point_y;
176:
177:    public int x {
178:       get { return point_x; }
179:       set { if ( value < 0 )
180:                point_x = 0;
181:             else
182:                point_x = value;  }
183:    }
184:    public int y {
185:       get { return point_y; }
186:       set { if ( value < 0 )
187:                point_y = 0;
188:             else
189:                point_y = value;  }
190:    }
191:
192:    public point()
193:    {
194:       x = 0;
195:       y = 0;
196:    }
197: }
198:
199: //-----------------------------------------------------------
200: /// <summary>
201: /// This class encapsulates line functionality
202: /// <see>point</see>
203: /// </summary>
204: class line
205: {
206:    private point lineStart;
207:    private point lineEnd;
208:
209:    public point start {
210:       get { return lineStart; }
211:       set { if ( value.x < 0 )
```

```
212:                lineStart.x = 0;
213:             else
214:                lineStart.x = value.x;
215:             if ( value.y < 0 )
216:                lineStart.y = 0;
217:             else
218:                lineStart.y = value.y;
219:          }
220:    }
221:    public point end {
222:       get { return lineEnd; }
223:       set { if ( value.x < 0 )
224:                lineEnd.x = 0;
225:             else
226:                lineEnd.x = value.x;
227:             if ( value.y < 0 )
228:                lineEnd.y = 0;
229:             else
230:                lineEnd.y = value.y;
231:          }
232:    }
233:
234:    public double length()
235:    {
236:       int x_diff;
237:       int y_diff;
238:       double length;
239:
240:       x_diff = end.x - start.x;
241:       y_diff = end.y - start.y;
242:
243:       length = (double) Math.Sqrt((x_diff * x_diff) + (y_diff * y_diff));
244:       return (length);
245:    }
246:
247:    public void DisplayInfo()
248:    {
249:        Console.WriteLine("\n\n-------------------------");
250:        Console.WriteLine("      Line stats:");
251:        Console.WriteLine("-------------------------");
252:        Console.WriteLine(" Length:     {0:f3}", length());
253:        Console.WriteLine(" Start Point: ({0},{1})", start.x, start.y);
254:        Console.WriteLine(" End Point:   ({0},{1})", end.x, end.y);
255:        Console.WriteLine("-------------------------\n");
256:    }
257:
258:    public line()
259:    {
260:       lineStart = new point();
261:       lineEnd = new point();
```

```
262:    }
263: }
264:
265: //-----------------------------------------------------------
266: /// <summary>
267: /// This class encapsulates square functionality
268: /// <see>line</see>
269: /// </summary>
270: class square
271: {
272:    private line squareHeight;
273:    private line squareWidth;
274:
275:    public line height {
276:       get { return squareHeight; }
277:       set {  squareHeight.start.y = value.start.y;
278:              squareHeight.end.y = value.end.y;
279:           }
280:    }
281:    public line width {
282:       get { return squareWidth; }
283:       set {  squareWidth.start.x = value.start.x;
284:              squareWidth.end.x = value.end.x;
285:           }
286:    }
287:
288:    public double area()
289:    {
290:       double total;
291:
292:       total = (width.length() * height.length());
293:       return (total);
294:    }
295:
296:    public double border()
297:    {
298:       double total;
299:
300:       total = ((2 * width.length()) + (2 * (height.length())));
301:       return (total);
302:    }
303:
304:    public void DisplayInfo()
305:    {
306:        Console.WriteLine("\n\n-------------------------");
307:        Console.WriteLine("    Square stats:");
308:        Console.WriteLine("-------------------------");
309:        Console.WriteLine(" Area:        {0:f3}", area());
310:        Console.WriteLine(" Border:      {0:f3}", border());
311:        Console.WriteLine(" WIDTH Points: ({0},{1}) to ({2},{3})",
```

```
312:                          width.start.x, width.start.y, width.end.x, width.end.y);
313:       Console.WriteLine("      Length: {0:f3}", width.length());
314:       Console.WriteLine(" HEIGHT Points: ({0},{1}) to ({2},{3})",
315:                                    height.start.x, height.start.y, height.end.x,
height.end.y);
316:       Console.WriteLine("      Length: {0:f3}", height.length());
317:
318:       Console.WriteLine("-------------------------\n");
319:    }
320:
321:    public square()
322:    {
323:       squareHeight = new line();
324:       squareWidth = new line();
325:
326:       width.start.x = 0;
327:       width.start.x = 0;
328:       width.end.x = 0;
329:       width.end.y = 0;
330:       height.start.x = 0;
331:       height.start.y = 0;
332:       height.end.x = 0;
333:       height.end.y = 0;
334:    }
335: }
336:
337: //---------------------------------------------------------
338: /// <summary>
339: /// This class encapsulates circle functionality
340: /// <see>line</see>
341: /// </summary>
342: class circle
343: {
344:    private point circleCenter;
345:    private long circleRadius;
346:
347:    public point center {
348:       get { return circleCenter; }
349:       set {  circleCenter.x = value.x;
350:            circleCenter.y = value.y;
351:          }
352:    }
353:    public long radius {
354:       get { return circleRadius; }
355:       set {  circleRadius = value; }
356:    }
357:
358:    public double area()
359:    {
360:       double total;
```

```
361:
362:        total = 3.14159 * radius * radius;
363:        return (total);
364:    }
365:
366:    public double circumference()
367:    {
368:        double total;
369:
370:        total = 2 * 3.14159 * radius;
371:        return (total);
372:    }
373:
374:    public void DisplayInfo()
375:    {
376:        Console.WriteLine("\n\n--------------------------");
377:        Console.WriteLine("    Circle stats:");
378:        Console.WriteLine("--------------------------");
379:        Console.WriteLine(" Area:        {0:f3}", area());
380:        Console.WriteLine(" Circumference: {0:f3}", circumference());
381:        Console.WriteLine(" Center Points: ({0},{1})", center.x, center.y);
382:        Console.WriteLine(" Radius:      {0:f3}", radius);
383:        Console.WriteLine("--------------------------\n");
384:    }
385:
386:    public circle()
387:    {
388:        circleCenter = new point();
389:
390:        center.x = 0;
391:        center.y = 0;
392:        radius = 0;
393:    }
394: }
395: //---------------------- End of Listing -------------------------
```

运行该程序时，将出现如下所示的菜单：

```
          Menu
============================
A - Working with Lines
B - Workin9 with Circles
C - Working with Squares
D - Working with Triangles
Q - Quit
============================

Enter Choice:
```

如果您输入的不是上述菜单中指定的字母，则将出现下面的消息：

```
          Menu
==========================

A - Working with Lines
B - Working with Circles
C - Working with Squares
D - Working with Triangles
Q - Quit

==========================

Enter Choice: g

   g is not valid <--
```

然后再次显示菜单，选择有效的选项将得到与下面类似的结果（这是输入 c 得到的结果）：

```
Enter Choice: c
----------------------------
       Square 8tats:
----------------------------
Area:          54. 0 00
Border:        3 0. 00 0
WIDTH Points:  (1, 0) to (10, 0)

      Length: 9. 0 00
HEIGHT Points: (0, 2) to (0, 8)
      Length: 6. 000
----------------------------

Press <ENTER> to continue...
```

XML 文档

正如前面指出的，使用该程序清单可以生成 XML 文档。下面是使用编译器选项/doc 生成的 XML 文档的内容：

```
< ?xml version= "1.0" ? >
< doc >
      < assembly >
      < name >wr01 < /name >
      < /assembly>
      < members >
      < member name= "M:WR01App.Main" >
         < summary>
         Main () routine that starts the application
         < /summary>
         < /member >
         < member name= "M:WR01App.DisplayMenu" >
               < summary >
               Displays a menu of choices.
               < /summary >
```

```
    </member>
    <member name="M: WR01App.GetMenuChoice" >
          < summary >
          Gets a choice from the user and verifies that it is valid.
          Returns a numeric value to indicate which selection was made.
          < /summary>
    < /member >
    < member name= "M: WR01App. WorkWithLine" >
          <summary>
          Method to perform code for Working with Line .
          < /summary>
    < /member>
    <member name= "M: WR01App. WorkWithCircle" >
          < summary >
          Method to perform code for Working with Circles.
          < /summary>
    < /member>
    <member name= "M: WR01App. WorkWithSquare" >
          < summary>
          Method to perform code for Working with Squares.
          < /summary>
    < /member >
    <member name= "M: WR01App. WorkWithTriangle" >
          < summary >
          Method to perform code for Working with Triangles.
          < /summary>
    < /member >
    <member name= "T: point" >
          < summary>
          This is a point class. It is for storing and
          working with an (x, y) value.
          < /summary>
    < / member >
    <member name= "T: line" >
          < summary>
          This class encapsulates line functionality
          < see >point < /see >
          < / summary >
    < /member >
    <member name= "T: square" >
          < summary>
          This class encapsulates square functionality
          < see > line< /see >
          < /summary >
    < /member >
<member name= "T: circle" >
    < summary >
    This class encapsulates circle functionality
    < see > line < /see >
    < /summary >
```

```
< /member >
        < /members >
< /doc>
```

代码概览

介绍程序清单的输出和 XML 文档后，我们来看一下代码。

该程序清单中有几项内容值得注意。首先，第 9 行将名称空间 System 包含到程序清单中。正如第 6 天介绍的，这意味着当您使用名称空间 System 内的东西时，不用指定 System；这些内容包括诸如 Console 方法等。该程序清单中声明了五个类：

- 第 11 ~ 65 行的 WR01App 类；
- 第 172 ~ 197 行的 point 类；
- 第 204 ~ 262 行的 line 类；
- 第 270 ~ 335 行的 square 类；
- 第 342 ~ 394 行的 circle 类。

注意： 在下一周您将知道，对于声明Point而言，有一种比使用类更好的方法——使用关键字struct。

line、square和circle类相似　Point类用于帮助组织其他类。

Main 方法

仔细查看该程序清单，您将知道该程序将从 Main 方法所在的第 16 行开始执行，该方法是在 WR01App 类中声明的。该方法使用一个 do...while 语句不断地处理菜单，直到用户选择合适的选项为止。菜单是通过调用另一个方法 GetMenuChioice 来显示的，然后根据该方法返回的值，执行多个不同例程中的一个。第 24 ~ 37 行的 switch 语句用于将程序流程转到正确的语句。

第 39 ~ 43 行使用一个 if 语句来检查用户选择的菜单选项。如果 menuChoice 为 0，则说明用户要退出。如果为其他的值，则在屏幕上显示信息。为在重新显示菜单之前，暂停程序的运行，加入了第 41 行和 42 行。第 41 行向用户显示一条消息，指出按 Enter 键继续。第 42 行使用 Console.ReadLine 方法来等待用户按下 Enter 键。对于用户在按 Enter 键之前输入的文本，程序将忽略它们。用户按 Enter 键继续后，while 语句的条件将被检查。如果 menuChoice 不为 0，则执行 do 语句，从而再次显示菜单，重复前面的处理过程。

来看一下 switch 语句，其中前面的四个 case 都分别执行一个方法，这些访问 WR01App 类的后边。如果 menuChoice 不是 1 ~ 4 之间的值，则 switch 语句中的 default（第 35 行）被执行，即打印一条错误消息。

GetMenuChoice 方法

第 22 行调用了方法 GetMenuChoice。该方法位于第 67 ~ 107 行，它显示菜单，并取得用户的选择。第 75 行调用另一个方法来实际显示菜单。显示菜单后，第 77 行使用方法 Console.ReadLine 来取得用户的选择，并将其赋给字符串变量 buf。

然后使用一个 switch 语句将该选项转换为数字值，并将其返回给调用方法。并非一定要进行这种转换，该方法的目的是取得选择的菜单选项，对于每个菜单选项，都有两种方式可以选择它。Swithch 语句接受其中的每种方式，并将其转换为一个正确的选项。完成这种工作的方式有很多

另外，您也可以返回字符值而不是数字值，这里需要指出的一点是，可以将获取被选择的菜单

选项的功能作为一个独立的方法，这样，您可以以任何方法来获知用户作出的选择，只要返回一组一致的最终选择值即可。您可以使用一个返回 0～4 的值的方法来替换该方法，而其他代码的运行方式不变。

菜单选项

四个菜单选项中的每一个调用一个方法。第 112～122 行包含方法 WorkWithLine，该方法声明一个对象，设置其初始值，最后调用该对象的一个方法来显示关于该对象的信息。方法 WorkWithSquare 和 WorkWithCircle 的运行方式与此相同。WorkWithTriangle 方法没有包含代码，而是留给您来完成。

point 类

point 类是在第 172～197 行定义的，它包含两个私有的数据成员：point_x 和 point_y。由于这些数据成员是私有的，因此 point 类外的代码不会访问它们。第 177～190 行使用关键字 get 和 set 声明了属性，这些属性为用户取得 point 的数据提供了入口。

第 192 行为 point 类声明了构造函数，该构造函数将数据成员的值设置为 0，这里使用属性来初始化这些值。

如果想改变 point 值的存储方式，可以修改属性。

line 类

line 类与 point 类类似。第 206～207 行声明了数据成员，这里的数据成员为 point 对象。第 258～263 行的构造函数为 line 类实例化两个 point 对象。注意，在该构造函数中，没有设置默认值。不需要这样做，因为当每个 point 类被实例化时，其构造函数将设置默认值。

line 类还包含其他可被调用的方法，这些方法的代码简单易懂。

其他类

该程序中的其他类与 line 类类似，您可以自行查看它们的代码。

注意： 您应该理解该程序清单中的所有代码，如果有什么地方不能理解，应回过头去复习相应的课程。以后您将学习如何对该程序清单进行改进。

第二周课程

您已学完了第 1 周的课程，还有两周的课程需要学习。本周将学习余下的大部分 C#核心概念——不是全部，而是大部分。经过本周的学习后，您将从零开始掌握构建基本的 C#程序所需的工具。

第 8 天课程“高级数据存储方式：结构、枚举和数组”将介绍其他存储数据的方式，包括结构、枚举和数组，其中结构类似于类。

第 9 天课程“关于方法的高级主题”将扩展第 7 和 8 天介绍的知识，介绍如何重载方法以及如何使用参数数目可变的方法。另外，还将介绍有关作用域方面的知识，让您能够限制对数据和其他类型成员的访问。您将学习关键字 static 以及如何创建不能用于创建对象的类。

创建程序时，避免让用户莫名其妙至关重要。第 10 天课程“处理异常”将介绍如何在程序中加入异常处理功能。异常处理技术是一种在问题导致程序崩溃之前捕获并防止它的结构化方式。

面向对象编程的关键之一是继承。第 11 天课程“继承”将介绍如何将继承用于您创建的类（或别人创建的类）。在这里，您将学习几个新的关键字，其中包括 sealed、is 和 as。

第 12 天课程“输入和输出”将复习一些稍微不那么重要的内容，让您稍事休息。这里将重点介绍如何在控制台上显示信息以及从控制台取得信息。您将学习如何格式化数据，使之更为有用。这里包含大量的表格，供您以后参考。

第 13 天课程“接口”介绍了理解 C#强大功能的另一个核心主题，这里将扩展您已学习到的关于类、结构和继承方面的知识，您将学习如何使用接口将多个特性组合到同一个类中。

最后，第 14 天课程“索引器、代表和事件”将重点介绍三个有趣的主题——索引器、代表和事件。您将学习如何将索引表示法用于类中的数据。另外，还将学习有关代表和事件的知识，让您动态地执行方法以及进行事件编程。事件是编写 Windows 应用程序的关键。

阅读完本周的课程后，您将学到许多 C#编程的核心概念。您将发现，您能理解大多数 C#程序中的大部分核心概念——不是全部，而是大部分。

第 8 天课程
高级数据存储方式：结构、枚举和数组

您已经学习过基本数据类型和类。C#还提供了其他一些在程序中存储信息的方式，今天的课程将介绍其中的几种，包括结构、枚举和数组。具体地说，今天将介绍以下内容：

- 如何在结构中存储值；
- 结构与类的异同；
- 枚举是什么？如何使用它来使程序更容易理解？
- 如何声明数组并使用它来存储大量数据类型相同的值？
- 使用关键字 foreach 来操纵数组。

8.1 结 构

结构是 C#提供的一种数据类型，它与类类似。和类一样，结构也可以包含数据和方法定义，还可以包含构造函数、常量、字段、方法、属性、索引器（indexer）、运算符和嵌套类型。

8.1.1 结构与类

虽然结构和类之间有很多相似性，但它们之间有一主要的区别和一些细微的区别。结构和类的主要区别在于存储和访问方式，结构是一种值数据类型，而类是一种引用数据类型。

虽然值数据类型和引用数据类型之间的差别已经在本书前面介绍过，但有必要重申，以便您能完全理解这种差别。值数据类型存储的是数据类型的名称指定的位置处的值；而引用数据类型存储的是信息的存储位置。图 8.1 再次给出了第 7 天中的一个图，该图说明了值数据类型和引用数据类型之间的差别。

从图 8.1 可以知道，引用变量实际上比存储值变量更复杂，但这种复杂性是由编译器负责处理的。虽然按引用存储信息有其优点，但要占用更多的内存。如果存储的信息量较少，则按引用存储带来的利将小于额外占用内存带来的弊

结构是按值存储的，没有引用带来的内存开销，因此在处理少量数据时优于类。

处理大量数据时，诸如类等引用类型是一种更合适的存储方式，特别是在传递数据给方法时。对于引用变量值，只传递引用，而不是整个数据值，而对于诸如结构等值变量，将被复制并传递给

方法。这种复制将带来额外的开销，如果结构很大，将降低程序的运行速度。

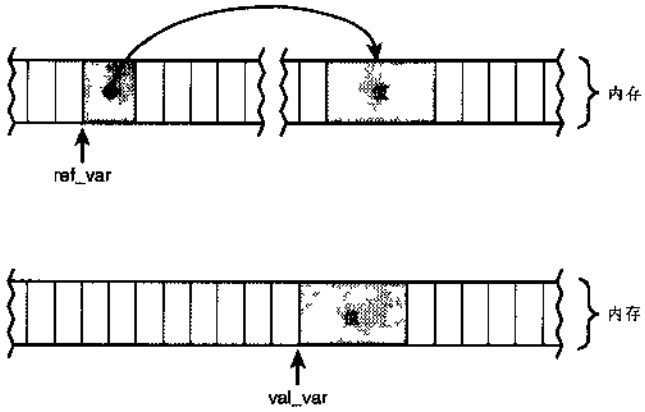


图 8.1 按值存储和按引用存储

一个通用的经验规则是，在决定使用类还是结构时，如果数据成员占用的内存不超过 16 个字节，则使用结构；否则，则应考虑您将如何使用这些数据。

8.1.2 结构成员

声明结构成员的方法与类成员相同，下面是一个用于存储点的结构：

```
struct point
{
   public int x;
   public int y;
}
```

这与前几天介绍的类类似，唯一的差别在于使用的是关键字 struct，而不是 class。您也可以在程序清单中以使用类的方式使用结构，程序清单 8.1 使用了该 point 结构。

程序清单 8.1 point.cs：使用 point 结构

```
 1: // point.cs- A structure with two data members
 2: //-------------------------------------------------------------
 3:
 4: struct point
 5: {
 6:    public int x;
 7:    public int y;
 8: }
 9:
10: class pointApp
11: {
12:   public static void Main()
13:   {
14:     point starting = new point();
15:     point ending = new point();
16:
17:     starting.x = 1;
```

```
18:       starting.y = 4;
19:       ending.x = 10;
20:       ending.y = 11;
21:
22:       System.Console.WriteLine("Point 1: ({0},{1})",
23:                                starting.x, starting.y);
24:       System.Console.WriteLine("Point 2: ({0},{1})",
25:                                ending.x, ending.y);
26:    }
27: }
```

该程序清单的输出如下：

```
Point 1: (1,4)
Point 2: (10,11)
```

分析：使用结构和使用类之间的主要差别在于，定义结构时使用的是关键字struct（如第4行所示）。事实上，您可以将关键字struct替换为class，该程序清单仍将能够运行。正如前面指出的，结构和类的主要差别在于它们在内存中的存储方式。图8.2说明了在内存中，point类对象starting和point结构实例是如何存储的。

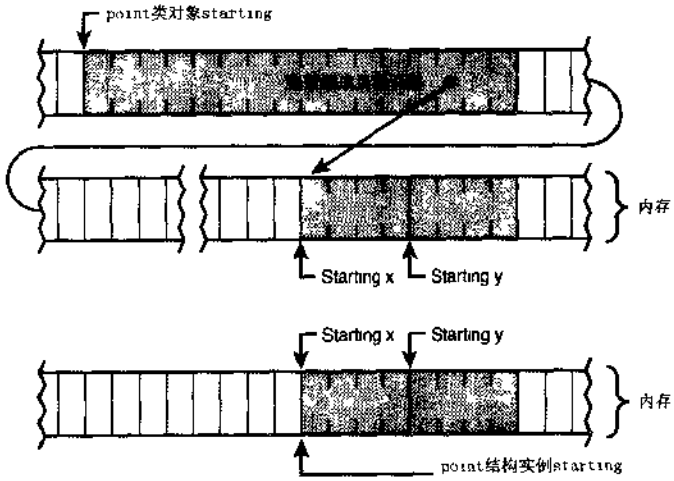


图 8.2　将点作为结构和类存储在内存中的情况

另外，访问结构成员的方式与类成员相同，即使用由成员运算符（句点）连接的结构实例名称和数据成员名称。例如，第 17 行访问 point 结构实例 starting 的数据成员 x。

声明结构实例可以更简单，即可以不使用关键字 new，这意味着第 14 行和 15 行可以修改成如下所示：

```
14:       point starting;
15:       point ending;
```

做上述修改后，重新编译该程序清单，它将能够通过编译，并能运行。如果就此将关键字 struct 改为 class，则编译时将出现以下错误：

```
point.cs(17,7): error CS0165: Use of unassigned local variable 'starting'
```

```
point.cs(19,7): error CS0165: Use of unassigned local variable 'ending'
```

8.1.3 结构的嵌套

和类一样，结构中可以包含其他数据类型，包括其他结构。程序清单 8.2 演示了一个包含两个 point 结构的 line 结构。

程序清单 8.2 line.cs：Line 结构

```
1:  // line.cs- A line structure which contains point structures.
2:  //-------------------------------------------------------
3:
4:  struct point
5:  {
6:     public int x;
7:     public int y;
8:  }
9:
10: struct line
11: {
12:     public point starting;
13:     public point ending;
14: }
15:
16: class lineApp
17: {
18:    public static void Main()
19:    {
20:       line myLine;
21:
22:       myLine.starting.x = 1;
23:       myLine.starting.y = 4;
24:       myLine.ending.x = 10;
25:       myLine.ending.y = 11;
26:
27:       System.Console.WriteLine("Point 1: ({0},{1})",
28:                                myLine.starting.x, myLine.starting.y);
29:       System.Console.WriteLine("Point 2: ({0},{1})",
30:                                myLine.ending.x, myLine.ending.y);
31:    }
32: }
```

该程序清单的输出如下：

```
Point 1: (1,4)
Point 2: (10,11)
```

分析： line结构的建立方式与前一天中的line类相似，主要差别在于实例化line结构时，将给它分配内存并直接存储它。

在该程序清单中，point 结构是在第 4～8 行声明的，第 12 行和 13 行声明了两个公有 point 结构

实例，其中的每个实例都有其坐标值 x 和 y。

第 22～25 行设置了 line 结构的各个值，可以按结构层次来访问嵌套结构的值，使用成员运算符（句点）将结构层次连接起来。第 22 行访问的是 line 结构 myLine 的 point 结构 starting 的数据成员 x。图 8.3 说明了 line 的层次。

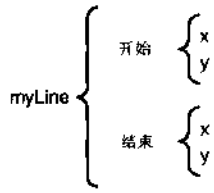


图 8.3　line 结构的层次

8.1.4　结构方法

和类一样，结构也可以包含方法和属性，它们的声明方式与类中相同，这包括使用与类中相同的限定符和属性（attribute）。您可以重载结构方法、传递值以及返回值。程序清单 8.3 演示了包含方法 length 的 line 结构。

程序清单 8.3　Line2.cs：在结构中添加方法

```
1:  // Line2.cs- Adding methods to a structure.
2:  //---------------------------------------------------------
3:
4:  struct point
5:  {
6:     public int x;
7:     public int y;
8:  }
9:
10: struct line
11: {
12:    public point starting;
13:    public point ending;
14:
15:    public double length()
16:    {
17:       double len = 0;
18:       len = System.Math.Sqrt( (ending.x - starting.x)*(ending.x - starting.x) +
19:                               (ending.y - starting.y)*(ending.y - starting.y));
20:       return len;
21:    }
22: }
23:
24: class lineApp
25: {
26:   public static void Main()
27:   {
28:      line myLine;
29:
```

```
30:       myLine.starting.x = 1;
31:       myLine.starting.y = 4;
32:       myLine.ending.x = 10;
33:       myLine.ending.y = 11;
34:
35:       System.Console.WriteLine("Point 1: ({0},{1})",
36:                                myLine.starting.x, myLine.starting.y);
37:       System.Console.WriteLine("Point 2: ({0},{1})",
38:                                myLine.ending.x, myLine.ending.y);
39:       System.Console.WriteLine("Length of line from Point 1 to Point 2: {0}",
40:                                myLine.length());
41:    }
42: }
```

该程序清单的输出如下：

```
Point 1: (1,4)
Point 2: (10,11)
Length of line from Point 1 to Point 2: 11.4017542509914
```

分析：该程序清单中加入了本书前面介绍过的length方法。该方法是在line结构中声明的，如第15～21行所示。与以前相同，该方法使用line结构的数据成员来计算线段的长度。然后将结果赋给变量len，并将其作为double值返回，如第20行所示。

lineApp 类调用该方法，并使用 Cosole.WriteLine 方法将返回的值显示出来，如第 39 行和 40 行所示。

虽然 line 结构只有一个方法，但您可以为它创建多个方法和属性，也可以重载这些方法。

8.1.5 结构的构造函数

除了可以包含常规方法外，结构还可以有构造函数。与类不同的是，为结构声明构造函数时，必须包含参数，不能为结构声明一个不包含任何参数的构造函数。程序清单 8.4 中包含一个带构造函数的 point 结构。

程序清单 8.4 point2.cs：包含构造函数的 point 结构

```
 1: // point2.cs- A structure with two data members
 2: //-------------------------------------------------------
 3:
 4: struct point
 5: {
 6:    public int x;
 7:    public int y;
 8:
 9:    public point(int x, int y)
10:    {
11:       this.x = x;
12:       this.y = y;
13:    }
14: //   public point()
15: //   {
```

```
16: //      this.x = 0;
17: //      this.y = 0;
18: //   }
19: }
20:
21: class pointApp
22: {
23:   public static void Main()
24:   {
25:     point point1 = new point();
26:     point point2 = new point(8, 8);
27:
28:     point1.x = 1;
29:     point1.y = 4;
30:
31:     System.Console.WriteLine("Point 1: ({0},{1})",
32:                              point1.x, point1.y);
33:     System.Console.WriteLine("Point 2: ({0},{1})",
34:                              point2.x, point2.y);
35:   }
36: }
```

该程序清单 的输出如下：

```
Point 1: (1,4)
Point 2: (10,11)
```

分析：结构和类的差别是不能为结构声明不包含任何参数的构造函数。该程序清单中包含一个不带任何参数的构造函数，但被注释掉了。如果删除这些行中的单行注释标记，并重新编译该程序清单，则将出现如下所示的错误：

```
point2.cs(14,12): error CS0568: Structs cannot explicit parameterless constructors
```

可以声明带参数的构造函数，第 9～13 行声明了一个对点坐标进行初始化的构造函数。point 结构的 x 和 y 的值被设置为传递给构造函数的值。为区分传入的 x、y 和结构实例的变量 x、y，使用了关键字 this，如第 11 行和 12 行所示。

第 25 行使用常规的方法实例化结构 point，也可以使用下面的代码来实例化结构实例 point1：

```
point point1;
```

即不包含 new 关键字和不带参数的构造函数调用。第 26 行使用了带参数的构造函数。

结构的构造函数必须初始化结构的数据成员。调用结构的默认构造函数（不带参数）时，将自动使用默认值初始化每个数据成员。通常，数据成员被初始化为 0，如果调用的是您声明的构造函数，而不是默认的构造函数，则您必须在构造函数中初始化所有的数据成员。

警告：虽然使用默认构造函数时，可以省略关键字new，但使用参数来实例化结构时，不能省略。将第26行修改为如下所示，将出错：

```
point point2(8,8);
```

8.1.6 结构的析构函数

类可以有析构函数。前面介绍过，虽然您可以使用类析构函数，但它们是不可靠的。结构不能有析构函数，如果您试图添加，将出现编译错误。

8.2 枚 举

在 C#中可以使用的另一种数据类型是枚举。枚举让您只能够创建有限的几个值的变量，例如每周只有七天，不将它们称为 1、2、3 等，而将其称为 Day.Monday、Day.Tuesday 和 Day.Wednesday 等，更清楚得多。对于可以是开或关的触发器，可以不使用诸如 0 和 1 等值，而是使用 Toggle.On 和 Toggle.Off。

枚举让您能够创建这样的值。枚举是使用关键字 enum 声明的，声明格式如下：

```
modifiers enum enumName
{
   enumMember1,
   enumMember2,
   ...
   enumMemberN
}
```

其中 modifiers 可以是关键字 new 或访问限定符（您熟悉的 public、private 或今天后面将介绍的 protected 和 internal）；enumName 是枚举的名称，可以是任何合法的标识符；enumMember1、enumMember2 到 enumMemberN 是枚举成员，包含描述性值。

下面声明了一个公有的 toggle 枚举：

```
public enum toggle
{
   On,
   Off
}
```

其中使用了关键字 enum，后面跟枚举的名称 toggle。该枚举有两个值：On 和 Off，它们之间由逗号分隔。要使用 toggle 枚举，可以声明一个数据类型为 toggle 的变量。例如下面声明了变量 myToggle：

```
toggle myToggle;
```

变量 myToggle 的合法取值有两个：On 和 Off。要使用这些值，可以使用枚举名和值名，并用成员运算符（句点）将它们连接起来。例如，myToggle 的值可以设置为 toggle.On 或 toggle.Off。使用 switch 语句可以检查这些不同的值。程序清单 8.5 演示了如何创建一个用于存储多个颜色值的枚举。

注意：默认情况下，在枚举变量被声明时，其值被设置为0。

程序清单 8.5：colors.cs：使用枚举

```
1: // color.cs- Using an enumeration
2: //         Note: Entering a nonnumeric number when running this
3: //               program will cause an exception to be thrown.
4: //---------------------------------------------------------------
5:
```

```
 6: using System;
 7:
 8: class myApp
 9: {
10:    enum Color
11:    {
12:       red,
13:       white,
14:       blue
15:    }
16:
17:    public static void Main()
18:    {
19:      string buffer;
20:      Color myColor;
21:
22:      Console.Write("Enter a value for a color: 0 = Red, 1 = White, 2 = Blue): ");
23:      buffer = Console.ReadLine();
24:
25:      myColor = (Color) Convert.ToInt32(buffer);
26:
27:      switch( myColor )
28:      {
29:        case Color.red:
30:           System.Console.WriteLine("\nSwitched to Red...");
31:           break;
32:        case Color.white:
33:           System.Console.WriteLine("\nSwitched to White...");
34:           break;
35:        case Color.blue:
36:           System.Console.WriteLine("\nSwitched to Blue...");
37:           break;
38:        default:
39:           System.Console.WriteLine("\nSwitched to default...");
40:           break;
41:      }
42:
43:      System.Console.WriteLine("\nColor is {0} ({1})", myColor, (int)myColor);
44:    }
45: }
```

该程序清单的输出如下：

```
Enter a value for a color: 0 = Red, 1 = White, 2 = Blue): 1

Switched to White...

Color is white (1)
Enter a value for a color: 0 = Red, 1 = White, 2 = Blue): 5
```

```
Switched to default...

Color is 5 (5)
```

分析：上述输出是运行两次该程序得到的。第一次运行时，输入的是1，它对应于white；第二次执行时，输入的是5，它不对应于任何颜色。

第 10～15 行声明了枚举 Color，它包含 3 个成员：red、white 和 blue。当该枚举被创建时，第一个成员的值被指定为 0；第二个成员的值被指定为 1；第三个成员的值被指定为 2。对于所有的枚举来说，第一个成员的值都是 0，其余的成员依次递增。

第 20 行使用该枚举来创建一个名为 myColor 的变量，该变量可以用来存储 Color 枚举的值之一。第 25 行给该枚举变量赋值。这里有必要对所赋的值做一些说明。第 22 行在屏幕上提示用户输入一个值，第 23 行使用 Console 类的 ReadLine 方法来读取用户输入的值，由于用户可能输入任何值，因此程序中可能出错。第 25 行假设用户输入的值可以被转换为一个整数，它使用 Convert 类的 ToInt32 方法来转换变量 buffer，buffer 变量包含用户输入的值。然后将转换结果强行转换为 Color 类型，并将其赋给变量 myColor。如果用于输入的不是数字，将出现运行错误，程序将终止。第 10 天将介绍一种妥善处理这种错误的方法，以便不显示运行错误，并继续执行程序。

第 27 行包含一个 switch 语句，根据 myColor 的值执行不同的分支。第 29～35 行的 case 语句中使用的不是字面值，而是枚举的值。因此 myColor 的值将与枚举的值进行匹配。该 switch 语句只是说明了如何根据枚举的不同值执行不同的分支而已。

有必要对第 43 行进行说明，该行打印两个值。第一个是 myColor 的值，您可能认为这是赋给变量的数字值，但实际上并非如此，而是枚举成员的名称。当 myColor 的值为 1 时，打印的是 white，而不是 1。如果要的是数字值，则必须显式地强制打印数字，在第 43 行通过强制转换实现了这一目的。

8.2.1 修改枚举的默认值

给枚举变量设置的默认值是 0。虽然给枚举变量指定的默认值为 0，但枚举中不一定非得包含一个值为 0 的成员。前面介绍过，枚举中第一个成员的值为 0，然后依次递增。实际上，可以修改这些默认值，例如，您常常希望第一个成员的值为 1，而不是 0。

要创建第一个成员的值为 1 的枚举，方法有两种。首先，可以在枚举的开始位置加上一个填充符值。如果希望第一个成员的值为 1，则这种方法很简单；但如果希望第一个成员的值是更大的数字，则这种方法便很糟糕。

第二种方法是设置枚举成员的值，可以将它们设置为字面值、其他枚举成员的值或计算得到的值。程序清单 8.6 并没有执行什么复杂的操作来设置枚举成员的值，而只是将第一个成员的值设置为 1（而不是 0）。第二个成员（February）的值为第一个成员的值加 1，这意味着无论是否像清单中那样显式地将其设置为 2，它的值都将是 2。

程序清单 8.6 bday.cs：设置枚举成员的数值型值

```
1: // bday.cs- Using an enumeration, setting default values
2: //---------------------------------------------------------
3:
4: using System;
5:
```

```
 6: public class myApp
 7: {
 8:    enum Month
 9:    {
10:       January = 1,
11:       February = 2,
12:       March = 3,
13:       April = 4,
14:       May = 5,
15:       June = 6,
16:       July = 7,
17:       August = 8,
18:       September = 9,
19:       October = 10,
20:       November = 11,
21:       December = 12
22:    }
23:
24:    struct birthday
25:    {
26:       public Month bmonth;
27:       public int   bday;
28:       puolic int   byear;
29:     }
30:
31:    public static void Main()
32:    {
33:      birthday MyBirthday;
34:
35:      MyBirthday.bmonth = Month.August;
36:      MyBirthday.bday = 11;
37:      MyBirthday.byear = 1981;   // This is a lie...
38:
39:      System.Console.WriteLine("My birthday is {0}, {1} {2}",
40:          MyBirthday.bmonth, MyBirthday.bday, MyBirthday.byear);
41:    }
42: }
```

该程序清单的输出如下：

```
My birthday is August, 11 1981
```

分析：该程序清单创建了一个名为Month的枚举类型，它包含12个月。这里将成员的值设置为1～12，而不是使用默认值0～11。由于成员的值将在前一个值的基础上加1，因此可以不显式地设置第二个成员以及以后的各个成员的值。这里之所以这样做，是为了更加清晰。您可以将这些值设置为其他的值，甚至可以使用公式。例如，可以将June的值设置为下面的公式：

```
May + 1
```

由于 May 被看作是 5，因此 June 的值将被设置为 6。

第 26 行使用枚举类型 Month 为结构声明了一个名为 bmonth 的公有数据成员。第 33 行使用结构 birthday 声明了一个名为 MyBirthday 的变量。然后在第 35～37 行给该结构实例的数据成员分别赋值。数据成员 bmonth 被赋给值 Month.August，也可以使用下面的强制转换方式将 bmonth 的值设置为 Month.August，但这样做将没有那么清晰：

```
MyBirthday.bmonth = (Month) 8;
```

同样，从第 39 行的打印结果可以知道，存储在 MyBirthday 中的值为 August，而不是数字 8。

8.2.2 修改枚举的底层类型

到目前为止的所有范例中，枚举的底层数据类型都是 int，实际上枚举可以包含数据类型为 byte、sbyte、int、uint、short、ushort、long 和 ulong 的值。如果未指定数据类型，则默认类型为 int。如果要在枚举中存储比 int 更大或更小的值，则可以将默认的底层类型修改为其他的类型。

修改默认类型的格式如下：

```
modifiers enum enumName: typeName {member(s)}
```

上述定义与前面介绍的相同，只是加入了冒号和 typeName。其中 typeName 可以是前面指出的任何一种类型。修改底层类型后，必须确保给成员指定数字型值的类型是有效的。

程序清单 8.7 使用了前面的 Color 枚举，但由于这些值较小，因此这里将枚举设置为使用 byte，以节省一些内存。

程序清单 8.7 color2.cs：显示随机的 byte 值

```
 1: // color2.cs- Using enumerations
 2: //---------------------------------------------------------
 3:
 4: using System;
 5:
 6: class myApp
 7: {
 8:    enum Color : byte
 9:    {
10:       red,
11:       white,
12:       blue
13:    }
14:
15:    public static void Main()
16:    {
17:      Color myColor;
18:      byte  roll;
19:
20:      System.Random rnd = new System.Random();
21:
22:      for ( int ctr = 0; ctr < 10; ctr++ )
23:      {
24:        roll = (byte) ((rnd.NextDouble() * 3 )); // random nbr form 0 to 2
25:        myColor = (Color) roll;
26:
```

```
27:         System.Console.WriteLine("Color is {0} ({1} of type {2})",
28:                 myColor, (byte) myColor,myColor.GetTypeCode());
29:       }
30:    }
31: }
```

该程序清单的输出如下：

```
Color is white (1 of type Byte)
Color is white (1 of type Byte)
Color is red (0 of type Byte)
Color is white (1 of type Byte)
Color is blue (2 of type Byte)
Color is red (0 of type Byte)
Color is red (0 of type Byte)
Color is red (0 of type Byte)
Color is blue (2 of type Byte)
Color is red (0 of type Byte)
```

注意：由于使用了随机数生成器，因此您运行该程序得到的输出可能不同。

分析：程序清单不仅仅是使用byte来声明枚举，稍后您将明白这一点。首先来看一下第8行，这里使用byte而不是int值来创建枚举Color，这是由于其中包含冒号和关键字byte。这意味着Color.red的值将是byte值0，Color.white将是byte值1，而Color.blue将是byte值2.

该程序清单的 Main 方法的功能与前一个程序清单不同，它使用了以前介绍过的随机逻辑。第 24 行生成一个 0～2 的随机数，并显式地将其强制转换为 byte 值，然后赋给变量 roll。在 18 行，roll 变量的类型被声明为 byte。第 25 行显式地将 roll 的值转换为 Color 类型，然后赋给变量 myColor。

第 27 行以后与前一个程序清单类似，使用 WriteLine 方法打印变量 myColor 的值（结果为 red、white 或 blue），然后通过显式强制转换为 byte 来打印其数字型值。但打印的第三个值是新的。

和 C#中的其他任何东西一样，枚举也是对象。因此，存在一些可用于枚举的内置方法。您将发现，其中最有用的一个是 GetTypeCode 方法，它返回变量的存储类型。对于 myColor，返回的类型为 byte，如输出中所示。如果在前两个清单中加入该参数，则结果将为 Int32。由于变量的类型是由运行阶段环境来确定的，因此返回的是.NET 框架数据类型，而不是 C#数据类型。

提示：有关用于枚举的其他方法，请参阅.NET框架文档中的Enum类。

应该	不应该
使用逗号，而不是分号来分隔枚举成员	填充符值不是枚举成员

8.3　使用数组存储数据

除了使用类和结构来将不同类型的相关信息存储在一起外，有时候您还需要存储多个数据类型相同的值。例如，银行可能需要跟踪每个月的结余，教师可能需要记录多次的考试成绩。

如果需要记录多个数据类型相同的条目，则最佳的方案是使用数组。要记录一年中每个月的结余，如果不使用数组，则需要创建 12 个变量来记录这些值：

```
decimal Jan_balance;
decimal Feb_balance;
```

```
decimal Mar_balance;
decimal Apr_balance;
decimal May_balance;
decimal Jun_balance;
decimal Jul_balance;
decimal Aug_balance;
decimal Sep_balance;
decimal Oct_balance;
decimal Nov_balance;
decimal Dec_balance;
```

要使用这些变量，首先要确定是哪个月，然后选择相应的变量。这需要多行代码，其中可能包含一条很长的 switch 语句，如下所示：

```
...
switch (month)
{
    case 1:  // do January stuff
         Jan_balance += new_amount;
         break;
    case 2:  // do February stuff
         Feb_balance += new_amount;
         break;
...
```

上面并没有列出整个 switch 语句，但足以说明确定月份并使用 12 个月结余值中的一个需要编写大量的代码。使用一个 decimal 数组来记录月结余的效率要高得多。

8.3.1 声明数组

数组是可以存储多个值的单个数据变量，其中所有值的数据类型都相同。在计算机内存中，这些数据元素是依次存储的，因此操纵它们以及在它们之间导航更为容易。

注意：*在程序清单依次声明多个变量并不意味着它们将在内存中被存储在一起。事实上，即使是依次声明的变量也可能被存储在内存中完全不同的地方。数组是包含多个元素的单个变量，因此在内存中，数组的值被存储在一起。*

要声明变量，可在声明该变量时在数据类型后面加上方括号。数组声明的基本格式如下：

```
datatype[] name;
```

其中，datatype 是将在数组中存储的数据类型；方括号指出您要声明一个数组；name 是该数组变量的名称。下面的代码声明了一个名为 balances 的数组，用于存储 decimal 值：

```
decimal [] balances;
```

上述声明创建了一个名为 balances 的数组，可用来存储 decimal 值，但并没有为存储变量预留空间。要预留空间，需要像创建其他对象那样，使用 new 关键字来实例化该变量。实例化数组时，必须指出该数组将存储多少个值。为此，可以使用的方法之一是，在实例化数组时在方括号内指出元素数目：

```
balances = new decimal[12]
```

也可以在声明数组的同时进行实例化，对 balances 而言，代码如下：

```
decimal[] balances = new decimal[12];
```

从上可以知道，实例化的格式如下：

```
new datatype [nbr_of_elements]
```

其中，datatype 是数组的数组类型，nbr_of_elements 是一个数值型值，指出该数组中存储的元素数目。从数组 balances 的实例化代码可以知道，它可以存储 12 个 decimal 值。

新术语：声明并实例化数组后，便可以使用它。数组中的条目被称为*元素*，其中的每个元素都可以通过*索引*进行访问。*索引*是一个标识数组中的偏移量（从而标识元素）的数字。

数组的第一个元素用索引 0 标识，这是因为第一个元素位于数组的开始位置，因此偏移量为 0。第二个元素的索引为 1，因为其偏移量为 1 个元素。最后一个元素的偏移量为数组长度减 1。例如，数组 balances 有 12 个元素，因此最后一个元素的索引为 11。

要访问数组中某个特定的元素，可使用后面跟方括号内带相应索引的数组名称。例如，要将数组 balances 的第一个元素的值设置为 1297.50，可以使用下面的代码：

```
balances[0] = 1297.50m;
```

将一个 decimal 值赋给数组 balances 的第三个元素的代码如下：

```
balances[2] = 1000m;
```

索引 2 用于访问第三个元素。程序清单 8.8 演示了如何使用数组 balances，图 8.4 则说明了元素和索引的概念，图中使用了一个更为简单的、包含三个字符的数组，其声明如下：

```
char[] intials = new char[3];
```

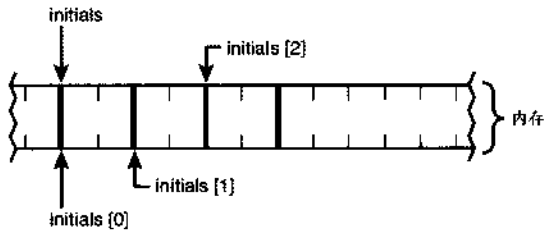


图 8.4　存储在内存中的数组及其索引

警告：一种常见的错误是忘记索引是从0而不是1开始的。在诸如Visual Basic等语言中，索引可以从1开始，但在大多数语言（包括C#）中，索引是从0开始的。

程序清单 8.8　balances.cs：使用数组

```
1: // balances.cs - Using a basic array
2: //--------------------------------------------------
3:
4: using System;
5:
6: public class myApp
```

```
 7: {
 8:    public static void Main()
 9:    {
10:       decimal[] balances = new decimal[12];
11:
12:       decimal ttl = 0m;
13:       System.Random rnd = new System.Random();
14:
15:       // Put random values from 0 to 100000 into balances array
16:
17:       for (int indx = 0; indx < 12; indx++ )
18:       {
19:          balances[indx] = (decimal) ((rnd.NextDouble() * 10000));
20:       }
21:
22:       //values are initialized in balances
23:
24:       for( int indx = 0; indx < 12; indx++ )
25:       {
26:          Console.WriteLine("Balance {0}: {1}", indx, balances[indx]);
27:          ttl += balances[indx];
28:       }
29:
30:       Console.WriteLine("==========================");
31:       Console.WriteLine("Total of Balances = {0}", ttl);
32:       Console.WriteLine("Average Balance = {0}", (ttl/12));
33:    }
34: }
```

该程序清单的输出如下：

```
Balance 0: 2276.50146106095
Balance 1: 4055.29556984794
Balance 2: 6192.0053633824
Balance 3: 2651.45477496621
Balance 4: 5885.39904257534
Balance 5: 2200.59107160223
Balance 6: 664.596651058922
Balance 7: 1079.63573237864
Balance 8: 2359.02580076783
Balance 9: 9690.85962031542
Balance 10: 934.673115114995
Balance 11: 7248.27192595614
==========================
Total of Balances = 45238.310129027017
Average Balance = 3771.54250645135085
```

分析：该程序清单演示了如何使用一个名为balances的基本数组。第10行将balances声明为一个decimal数组，并被实例化为包含12个元素。第13行创建了一个Random对象，用于创建存储在数组中

的随机数。将随机数赋给数组元素的操作是在第17～20行完成的，该for循环使用索引计数器indx从0循环到11，并在第19行将该计数器用作数组的索引。

给数组元素赋值后，第 24～28 行遍历该数组。从技术上讲，该循环是多余的，但通常情况下，您会从其他地方获得数组元素的值，而不是将随机数指定给它们。第二个 for 循环将数组 balances 的每个元素显示到屏幕上（第 26 行）。第 27 行将数组 balances 的每个元素的值加到总数 ttl 中。第 31 和 32 行则打印了一些关于这些随机结余的总结性信息。第 31 打印了结余总量，而第 32 行打印了平均结余。

与使用 12 个变量相比，使用 balances 数组的代码要简单得多。结合使用数组名和索引时，如 balances[2]，就像同类型的常规变量一样。

8.3.1.1　使用数组初始化数组元素

在声明和实例化数组的同时，可以初始化各个数组元素的值。为此，可以在变量声明后面指定数组元素的值。这些值包含在大括号内，之间用逗号分隔。要初始化数组 balance 的值，可以使用下面的代码：

```
decimal [] balances = new decimal[12] {1000.00m, 2000.00m, 3000.00m, 4000.00m, 5000m, 6000m,
0m, 0m, 9m, 0m, 0m, 12000m};
```

上述声明创建数组 balances，并预先给它赋值。第一个值 1000.00m 将赋给第一个元素 balances[0]；第二个值 2000.00m 将赋给第二个元素 balances[1]；依此类推。

有趣的是，以这种方式初始化数组的值，不需要在方括号内指定数组的大小。下面的语句与上述语句等效：

```
decimal [] balances = new decimal[] {1000.00m, 2000.00m, 3000.00m, 4000.00m, 5000m, 6000m,
0m, 0m, 9m, 0m, 0m, 12000m};
```

编译器自动将该数组定义为包含 12 个元素，因此初始化的是 12 个值。程序清单 8.9 创建并初始化了一个字符数组。

注意：如果在声明中指出了元素数目，则可以不初始化所有的元素。下面的代码行是合法的，数组将包含12个元素，开始两个元素的值被初始化为111：

```
decimal[] balances = new decimal[12] {111m, 111m};
```

但如果没有指出元素数目，则以后不能增加元素。在下面的声明中，balances数组只有两个元素，因此不能存储两个以上的元素

```
decimal[] balances = new decimal [] {111m, 111m};
```

程序清单 8.9　fname.cs：使用数组

```
 1: // fname.cs - Initializing an array
 2: //---------------------------------------------------------
 3:
 4: using System;
 5:
 6: public class myApp
 7: {
 8:    public static void Main()
 9:    {
10:       char [] name = new char[] {'B','r','a','d','l','e','y', (char) 0 };
11:
```

```
12:        Console.WriteLine("Display content of name array...");
13:
14:        int ctr = 0;
15:        while (name[ctr] != 0)
16:        {
17:           Console.Write("{0}", name[ctr]);
18:           ctr++;
19:        }
20:        Console.WriteLine("\n...Done.");
21:     }
22: }
```

该程序清单的输出如下：

```
Display content of name array...
Bradley
...Done.
```

分析：第10行创建、实例化并初始化了一个名为name的字符数组，该数组被实例化为包含8个元素。您知道它将包含8个元素，因为虽然没有专门指出，但在声明时将8个元素加入了该数组中

该程序清单还包含一些以前没有介绍过的功能，它将一个奇怪的值（ASCII 码为 0 的字符）赋给数组的最后一个元素，这个值用于指示数组的结尾。第 14 ~ 19 行创建了一个名为 ctr 的计数器，并将其用作索引。ctr 用于遍历数组的元素，直到找到一个 ASCII 码为 0 的字符。至此，while 语句的条件为 false，循环结束。这样可以防止您超越数组的结尾处。

8.3.2 多维数组

多维数组是一个由数组组成的数组，您甚至可以使用三维数组！这将使数组急剧地复杂化，因此建议您不要使用超过三维的数组。

由数组组成的数组通常被称为二维数组，因为它可以以平面的方式进行表示。要声明二维数组，只需对声明常规数组的方法进行扩展即可：

```
byte [,] scores = new byte[15, 30];
```

声明的第一部分增加了一个逗号，而第二部分则使用了由逗号分隔的两个数字。上述声明创建了一个包含 15 个元素的数组，其中的每个元素都是一个包含 30 个元素的数组。因此，数组 scores 总共可以存储 450 个数据类型为 byte 的值。

要声明一个存储几个字符的简单多维数组，可以使用下面的代码：

```
char[,] letters = new char[2,3]; //without initializing values
```

该声明创建一个名为 letters 的二维数组，它包含两个元素，其中每个元素都是一个包含三个字符元素的数组。可以在声明时初始化数组 letters 的元素：

```
char[,] letters = new char[2,3] {{'a', 'b', 'c'},{'X', 'Y', 'Z'}};
```

也可以分别初始化各个元素。要访问多维数组的元素，同样是使用索引。数组 letters 的第一个元素是 letters[0,0]（记住索引从 0 开始，而不是 1 开始）；第二个元素是 letters[0,1]，其值为字母 b。字母 X 是元素 letters[1,0]的值，因为位于第二个数组（偏移量为 1）的第一个元素（偏移量为 0）中。

要在声明之外初始化数组 letters，可以使用下面的代码：

```
letters[0,0] = 'a';
letters[0,1] = 'b';
letters[0,2] = 'c';
letters[1,0] = 'X';
letters[1,1] = 'Y';
letters[1,2] = 'Z';
```

8.3.3　创建包含的数组长度不同的数组

前一节假设二维数组中所有子数组的长度都相同，这样数组将呈矩形。如果要存储长度不同的数组，该如何办呢？请看下面的例子：

```
char [] [] myname = new char[3] [];
   myname[0] = new char[] {'B', 'r', 'd', 'l', 'e', 'y'};
   myname[1] = new char[] {'L', '.'};
   myname[2] = new char[] {'J', 'o', 'n', 'e', 's'};
```

数组 myname 是一个由数组组成的数组，包含三个长度各不相同的字符数组。由于它们的长度不同，因此使用它们的方式不同于前面的矩形数组。图 8.5 说明了数组 myname。

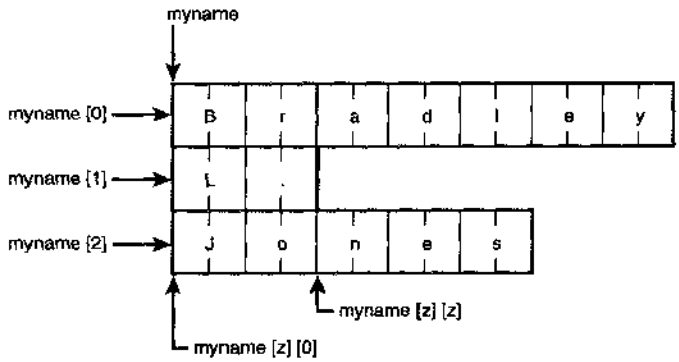


图 8.5　由长度各不相同的数组组成的数组

引用各个元素时，不是将用逗号分隔的索引放在同一个方括号内，而是将索引值放在不同的方括号内。例如，下面的代码行使用方法 WriteLine 打印作者姓和名的头一个字母：

```
System.Console.WriteLine("{0}{1}{2}", myname[0][0], myname[1][0], myname[2][0]);
```

以 myname[0,0]、myname[1,0]和 myname[2,0]的方式引用上述元素将出错。

注意：由长度相同的子数组组成的数组被称为矩形数组；由长度不同的子数组组成的数组被称为锯齿状数组（jagged）。从图8.5使可以明白为何被称为锯齿状数组。

如果您不想像前面那样在声明数组 myname 的同时初始化它，情况将如何呢？您知道该数组有三部分组成，因此第一维的长度为 3；但第二维的长度是多少呢？由于子数组的长度各不相同、因此必须执行多次实例化来创建出整个数组。首先，您声明包含子数组的数组：

```
char[][] myname = new char[3][];
```

上述代码声明了一个名为 myname 的、包含三个元素的数组，其中的每个元素都是存储字符的数组。完成上述声明后，必须实例化存储在 myname[]中的各个数组（图 8.5 说明了 myname 数组）：

```
myname[0] = new char[7]; //first array of seven elements
myname[1] = new char[2]; //second array of two elements
myname[2] = new char[5]; //third array of five elements
```

8.3.4 检查数组长度和边界

在演示锯齿形多维数组 myname（如程序清单 8.10 所示）之前，有必要对另一项内容进行介绍。每个数组都知道其长度，数组的长度被存储在一个名为 Length 的成员中。和 C#中的其他所有东西一样，数组也是对象。要知道数组的长度，可以像其他数据成员和对象那样使用数据成员 Length。对于一维数组 balance，可以通过 balance.Length 获知其长度。

在多维数组中，可以使用 Length 或数组的方法 GetLength( )来获知其子数组的长度。您通过传递了数组的索引号来指出要返回哪个子数组的长度。程序清单 8.10 演示了如何使用锯齿形数组的成员 Length。

程序清单 8.10 names.cs：使用锯齿形的二维数组

```
1:  // names.cs - Using a two-dimensional array
2:  //-------------------------------------------------------
3:
4:  using System;
5:
6:  public class myApp
7:  {
8:     public static void Main()
9:     {
10:       char[][] name = new char[3][];
11:
12:       name[0] = new char[7] {'B', 'r', 'a', 'd', 'l', 'e', 'y'};
13:       name[1] = new char[2] {'L', '.'};
14:       name[2] = new char[5] {'C', 'o', 'n', 'e', 's'};
15:
16:       Console.WriteLine("Display the sizes of the arrays...\n");
17:
18:       Console.WriteLine("Length of name array {0}", name.Length);
19:
20:       for( int ctr = 0; ctr < name.Length; ctr++)
21:          Console.WriteLine("Length of name[{0}] is {1}", ctr, name[ctr].Length);
22:
23:  //-------------------------------------------------------
24:
25:       Console.WriteLine("\n\nDisplaying the content of the name array...");
26:
27:       for( int ctr = 0; ctr < name.Length; ctr++)
28:       {
29:          Console.Write("\n"); // new line
30:          for( int ctr2 = 0; ctr2 < name[ctr].Length; ctr2++ )
31:          {
32:             Console.Write("{0}", name[ctr][ctr2]);
33:          }
```

```
34:       }
35:       Console.WriteLine("\n...Done displaying");
36:    }
37: }
```

该程序清单的输出如下：

```
Display the sizes of the arrays...

Length of name array 3
Length of name[0] is 7
Length of name[1] is 2
Length of name[2] is 5

Displaying the content of the name array...

Bradley
L.
Jones
...Done displaying
```

分析：来看看该程序清单的各个部分。第一部分是第10～14行。第10行声明了一个名为name的二维数组，它包含三个长度可能不同的字符数组。第12～14行分别实例化这三个数组，虽然其中的方括号中包含了数组的长度，但由于数组被初始化，因此可以不包含这些数字。但好的习惯是包含这些数字，并明确您的意图。

第二部分演示了数组成员 Length。第 18 行打印数组 name 的长度，您可能认为结果为 14，但实际为 3。成员 Length 指的是元素数目，name 数组有三个元素，其中的每个元素都是一个数组。

第 20 行将数组 name 的长度（这里为 3）作为上限，来遍历其中的每个组数。使用一个索引计数器打印每个数组的长度。从输出可以知道，这些值与声明的相同。

第三部分是第 27～34 行，它显示存储在各个 name 数组中的值。通过自动检查每个子数组的 Length 成员，而不是使用硬编码值，使得代码是动态的。如果在第 12～14 行修改数组的长度，则程序清单的其他部分仍管用。

8.3.5　在类和结构中使用数组

数组只不过是另一种可用于声明变量的类型而已。在可以使用其他数据类型的地方也可以创建和使用数组。这意味着可以在结构、类以及其他数据类型中使用数组。

注意：虽然今天使用的都是基本数据类型，但您几乎可以创建任何数据类型的数组。您可以使用类、结构或其他任何数据类型来创建数组。

8.3.6　使用 foreach 语句

现在是介绍关键字 foreach 的时候了。foreach 关键字使得使用数组更为简单，特别是遍历整个数组时。另外，它不是使用带下标的数组名来引用数组，而是使用一个简单的变量。foreach 语句的缺点在于，使用的简单变量是只读的——不能给它赋值。foreach 命令的格式如下：

```
foreach( datatype varname in arrayName)
{
   statements;
```

```
}
```

其中 datatype 是数组的数据类型，varname 是用于标识数组中各个元素的变量的名称；arrayname 是 foreach 要遍历的数组的名称。程序清单 8.11 演示了如何使用 foreach 来遍历一个名为 name 的数组。

程序清单 8.11　foreach1.cs：将 foreach 用于数组

```
 1: // foreach1.cs - Initializing an array
 2: //-----------------------------------------------------
 3:
 4: using System;
 5:
 6: public class myApp
 7: {
 8:    public static void Main()
 9:    {
10:       char[] name = new char[] {'B','r','a','d','l','e','y'};
11:
12:       Console.WriteLine("Display content of name array...");
13:
14:       foreach( char x in name )
15:       {
16:          Console.Write("{0}", x);
17:       }
18:
19:       Console.WriteLine("\n...Done.");
20:    }
21: }
```

该程序清单的输出如下：

```
Display content of name array...
Bradley
...Done
```

分析：该程序清单比前一个短，其重点在第14行，它使用关键字foreach遍历数组name。它遍历数组的每一个元素，然后结束。遍历时，它依次将各个元素称为x。在foreach语句的代码中，不需要使用array[index_ctr]，而是使用x。

8.4 总　结

今天的课程介绍了三种重要的数据类型：结构、枚举和数组。结构的工作原理与类类似，它们之间的重要差别在于，结构是值类型，而类是引用类型。枚举（使用关键字 enum 声明）对提供代码的可读性很有帮助。枚举让您能够创建一种取值范围由您控制的数据类型，同时可以给这些值指定更适用的名称。

今天讨论的最后一个主题是数组。您知道如何创建数组，还知道数组可以是多维的。多维数组的子数组的长度可以相同（矩形数组），也可以不同（锯齿形数组）。

最后介绍了 foreach 关键字，该关键字使得使用数组简单得多。

8.5 问与答

问：在结构和类之间，还有今天没有介绍过的其他差别吗？

答：有，还有几个今天没有介绍的其他差别。您知道，结构是按值存储的，而类是按引用存储的。另外，结构的构造函数必须包含参数，而且不能有析构函数。除此之外，结构是隐式密封的（sealed），这一概念将在讨论继承时介绍。

问：据说枚举可以和位字段一起使用，这是如何实现的？

答：这是一个比较高级的主题，本书不打算介绍。可以使用枚举来存储各位的值。可以通过这样来实现，使用 byte 成员，每个成员被设置为字节中某位对应的位置。相应的枚举如下：

```
enum Bits : byte
{
      first = 1,
      second = 2,
      third = 4,
      fourth = 8,
      fifth = 16,
      sixth = 32,
      seventh = 64,
      eighth = 128
}
```

然后使用这些预定义的值，通过位运算符来执行位运算。

问：枚举是值类型还是引用类型？

答：当变量被声明为枚举时，它将是值类型。值实际上被存储在枚举变量中。

问：可以将数组声明为多少维？

答：可以将数组的维数声明为比实际需要的多些。如果声明三维以上的数组，将出现两种问题之一：要么由于使用矩形数组而浪费大量的内存空间，要么代码更为复杂。几乎在所有的情况下，都可以找到更为简单的处理信息的方法，这些方法不需要三维以上的数组。

8.6 作　业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

8.6.1 小测验

1. 值数据类型和引用数据类型的差别何在？结构属于哪一种？
2. 结构和类的区别何在？
3. 结构的构造函数和类的构造函数之间有何差别（或者它们之间有差别吗）？
4. 哪个关键字用于声明枚举？
5. 在枚举中可以存储哪些数据类型？
6. 数组第一个元素的索引值是多少？
7. 如果访问数组元素时使用的索引值大于数组的元素数目，将出现什么情况？

8. 被声明为 myArray[4,3,2]的数组包含多少个元素，如果这是一个字符数组，将占用多少内存？

9. 如何知道数组的长度？

10. 判断正误（如果错误，请指出错在哪里）：foreach 的结构与 for 语句相同。

8.6.2 练习

1. 修改程序清单 8.3 中的结构 point 和 line，给其数据成员加上属性。

2. 选做题：修改 line 结构，使之包含一个静态数据值，它保存曾经存储过的最长的线段的长度。每当 Length 方法被调用时，都需检查并更新该静态变量。

3. 下面的代码片段有问题，您能够更正吗（假设 myArray 是一个 decimal 数组）？

```
foreach( decimal Element in myArray )
{
     System.Console.WriteLine("Element value is: {0}", Element);
     Element *= Element;
     System.Console.WriteLine("Element squared is: {0}", Element);
}
```

4. 为教师编写一个程序，该程序使用一个数组存储 30 个学生的考试成绩，并给各个数组元素指定一个 1～100 的随机值，然后计算平均成绩。

5. 修改练习 4 中创建的程序，以记录整个学期中 15 次考试的考试成绩。当然 30 个学生的考试成绩都要记录。并打印各次考试的平均成绩和各个学生的学期平均成绩。

6. 修改练习 5 创建的程序，以记录 5 年的全部考试成绩。使用一个三维数组来实现。

第 9 天课程
关于方法的高级主题

前 8 天介绍了大量的内容，今天将以这些知识为基础进一步介绍类方法。第 6 天和第 7 天介绍了如何将功能和数据封装到类中，今天将介绍的重要知识之一是如何提高类的灵活性。今天的课程包含以下内容：

- 如何给方法传递不同的参数；
- 如何重载方法；
- 方法的特征标（signature）；
- 再谈作用域；
- 创建自己的名称空间。

9.1 重载方法

多态是面向编程语言的特性之一。正如本书前面介绍的，多态指的是即使提供不同的选项，也能得到相同的结果。重载是多态的形式之一，前面在介绍多态时，使用了一个圆的例子。

在 C#中，最常使用重载的是方法。可以创建出对许多不同的值做出反应，并得到相同结果的方法。请看图 9.1。图中的黑盒子是计算圆面积的方法。

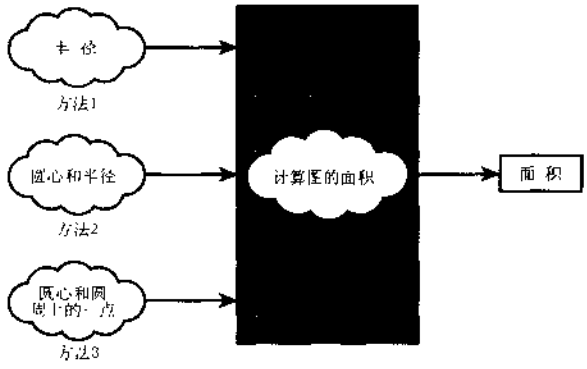


图 9.1　计算圆面积的黑盒子

以三种不同的方法来提供圆参数，都将得到正确的结果。第一种方法是只提供半径；第二种方

法是提供半径和圆心；第三种方法是提供圆心和圆周上的一点。三种方法都要求 area 方法返回圆的面积。

编写实现上述黑盒子功能的程序时，您可能创建三个不同的方法，并将这些方法分别称为 CalcArea、CalcAreaWithPoints、CalcAreaWithRadius 或无数个其他独一无二的名称之一。如果编程时使用的是诸如 C 语言等非面向对象的语言，则需要创建多个函数；但使用具备多态功能的面向对象语言时，有一种更简单的解决方案——方法重载。

9.1.1 重载函数

新术语：*方法重载*指的是创建多个名称相同的方法。其中的每个方法都在某些方面具备唯一性，以便编译器能够区别它们。程序清单 9.1 包含一个 Circle 类，其 Area 方法被重载，以便图 9.1 所示的每种调用方式都可行。

程序清单 9.1 circle.cs：通过方法重载来说明多态

```
1: // circle1.cs - polymorphic area method
2: //---------------------------------------------------------
3:
4: using System;
5:
6: public class Circle
7: {
8:    public int x;
9:    public int y;
10:   public double radius;
11:   private const float PI = 3.14159F;
12:
13:   public double Area()  // Uses values from data members
14:   {
15:      return Area(radius);
16:   }
17:
18:   public double Area( double rad )
19:   {
20:      double theArea;
21:      theArea = PI * rad * rad;
22:      Console.WriteLine(" The area for radius {0}) is {1}", rad, theArea);
23:      return theArea;
24:   }
25:
26:   public double Area(int x1, int y1, double rad)
27:   {
28:       return Area(rad);
29:   }
30:
31:   public double Area( int x1, int y1, int x2, int y2 )
32:   {
33:      int x_diff;
34:      int y_diff;
35:      double rad;
```

```
36:
37:        x_diff = x2 - x1;
38:        y_diff = y2 - y1;
39:
40:        rad = (double) Math.Sqrt((x_diff * x_diff) + (y_diff * y_diff));
41:
42:        return Area(rad);
43:     }
44:
45:     public Circle()
46:     {
47:        x = 0;
48:        y = 0;
49:        radius = 0.0;
50:     }
51: }
52:
53: class CircleApp
54: {
55:    public static void Main()
56:    {
57:       Circle myCircle = new Circle();
58:
59:       Console.WriteLine("Passing nothing...");
60:       myCircle.Area();
61:
62:       Console.WriteLine("\nPassing a radius of 3...");
63:       myCircle.Area( 3 );
64:
65:       Console.WriteLine("\nPassing a center of (2, 4) and a radius of 3...");
66:       myCircle.Area( 2, 4, 3 );
67:
68:       Console.WriteLine("\nPassing center of (2, 3) and a point of (5, 6)...");
69:       myCircle.Area( 2, 3, 4, 5 );
70:    }
71: }
```

该程序清单的输出如下：

```
Passing nothing...
   The area for radius (0) is 0

Passing a radius of 3...
   The area for radius (3) is 28.2743110656738

Passing a center of (2, 4) and a radius of 3...
   The area for radius (3) is 28.2743110656738

Passing center of (2, 3) and a point of (5, 6)...
    The area for radius (2.82842712474619) is 25.1327209472656
```

分析：首先来看看第60、63、66和69行，它们都调用myCircle对象的Area方法，但提供的参数数目不同。该程序仍然通过编译并正确运行！

这是通过方法重载实现的。Circle 类定义了四个 Area 方法，它们之间的区别在于传递的参数数目不同。第 13～16 行定义了一个不接受如何参数，但仍然返回一个 double 值的 Area 方法，其方法体调用接受一个参数的 Area 方法，并传递存储在数据成员 radius 中的半径值。

第 18～24 行定义了第二个 Area 方法，该方法接受一个 double 值，并将其作为圆的半径。第 21 行使用传入的半径值计算圆的面积。第 22 行将圆的半径和计算得到的面积打印到屏幕上。该方法的最后部分将面积返回给调用例程。

提示：在该程序清单中，对于PI使用的不是常量值，而是在第11行声明了一个固定变量。这样，当需要修改PI的值时，只要在一个地方进行修改即可，而不用对分布在应用程序各个地方的硬编码进行修改。以后维护起来也将更容易。

第 26～29 行是第三个 Area 方法的定义。该方法定义有些可笑，因为计算面积只需要半径即可。该方法将半径值 rad 传递给只需要半径的 Area 方法，而不是重复实现相同的功能。面积值将依次从各个方法返回，最后到达最初的调用者。

第 31～43 行是最复杂的 Area 方法，它接受圆心和圆周上的一点，半径将是这两点之间的直线距离（如图 9.2 所示）。第 40 行根据两点的位置计算半径的长度。然后将得到的结果传递给只需要半径的 Area 方法，由它来完成余下的工作。

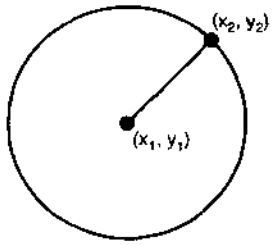


图 9.2　圆心和圆周上的一点

虽然这些方法都调用另一个 Area 方法，但并非一定要这样做。这些方法可以包含各自完全独立的编码。如何编写它们取决于您，重要的是，您可以创建多个名称相同的方法，它们根据不同的值执行操作。

警告：虽然可以在名称相同的各个方法中实现完全不同的功能，但这并不意味着应该这样做。因为它们的名称相同，因此最后的结果应该是类似的。

9.1.2　构造函数重载

除了可以重载常规方法外，还可以重载构造函数。重载的构造函数让您能够在创建对象的同时给它传递值。程序清单 9.2 包含一个其构造函数被重载的 Circle 类，该 Circle 类不同于程序清单 9.1 中的那个。

程序清单 9.2　circle1.cs：重载构造函数

```
1: // circle1.cs - A simple circle class with overloaded constructors
2: //---------------------------------------------------------------
3:
```

```
4:  using System;
5:
6:  public class Circle
7:  {
8:     public int x;
9:     public int y;
10:    public int radius;
11:    private const float PI = 3.14159F;
12:
13:    public double area()
14:    {
15:       double theArea;
16:       theArea = PI * radius * radius;
17:       return theArea;
18:    }
19:
20:    public double circumference()
21:    {
22:       double Circ;
23:       Circ = 2 * PI * radius;
24:       return Circ;
25:    }
26:
27:    public Circle()
28:    {
29:       x = 0;
30:       y = 0;
31:       radius = 0;
32:    }
33:
34:    public Circle( int r )
35:    {
36:       x = 0;
37:       y = 0;
38:       radius = r;
39:    }
40:
41:    public Circle ( int new_x, int new_y )
42:    {
43:       x = new_x;
44:       y = new_y;
45:       radius = 0;
46:    }
47:
48:    public Circle ( int new_x, int new_y, int r )
49:    {
50:       x = new_x;
51:       y = new_y;
52:       radius = r;
53:    }
```

```
54:
55:     public void print_circle_info()
56:     {
57:       Console.WriteLine("Circle: Center = ({0},{1})", x, y);
58:       Console.WriteLine("        Radius = {0}", radius);
59:       Console.WriteLine("        Area   = {0}", area());
60:       Console.WriteLine("        Circum = {0}", circumference());
61:     }
62: }
63:
64: class CircleApp
65: {
66:    public static void Main()
67:    {
68:       Circle first  = new Circle();
69:       Circle second = new Circle(4);
70:       Circle third  = new Circle(3,4);
71:       Circle fourth  = new Circle(1, 2, 5);
72:
73:       Console.WriteLine("\nFirst Circle:");
74:       first.print_circle_info();
75:
76:       Console.WriteLine("\nSecond Circle:");
77:       second.print_circle_info();
78:
79:       Console.WriteLine("\nThird Circle:");
80:       third.print_circle_info();
81:
82:       Console.WriteLine("\nFourth Circle:");
83:       fourth.print_circle_info();
84:    }
85: }
```

该程序清单的输出如下：

```
First Circle:
Circle: Center = (0,0)
      Radius = 0
      Area   = 0
      Circum = 0

Second Circle:
Circle: Center = (0,0)
      Radius = 4
      Area   = 50.26544
      Circum = 25.13272

Third Circle:
Circle: Center = (3,4)
      Radius = 0
```

```
        Area   = 0
        Circum = 0

Fourth Circle:
Circle: Center = (1,2)
        Radius = 5
        Area   = 78.53975
        Circum = 31.41590001106262
```

分析：该程序清单的重点是Circle类的构造函数。有多个构造函数，每一个接受的参数个数都不同，第27～32行定义了第一个构造函数，您在本书的前面见到过这样的构造函数。该构造函数声明为公有的，名称为类名，这完全符合第7天介绍的格式。

第 34～39 行是另外三个 Circle 构造函数的第一个，它接受一个表示半径的整数，这个值被作为类的半径字段。

第 41～46 行是第三个构造函数，它与其他构造函数的区别在于接受两个整数值——圆心的 x 坐标和 y 坐标。第 43～44 行将对象的值设置为这些值。

第四个也是最后一个 Circle 构造函数位于第 48～53 行，它接受三个值——半径以及圆心的 x 坐标和 y 坐标。

这些方法的声明方式和名称都相同，唯一的差别是介绍的参数数目不同。

CircleApp 类中的第 68～71 行使用了这些构造函数。第 68 行创建一个名叫 first 的新 Circle 对象，声明和创建的方式与以前介绍的创建对象方式相同。

第 69 行创建第二个对象的方式则不同，它不是使用 new Circle( )，而是包含添加了一个参数——4。创建该对象需要使用接受一个数字值（4）的构造函数，这正是从第 34 行开始的构造函数，该构造函数的格式与第 69 行的调用相同。

基于上述关于创建第二个对象的描述，可以很容易知道，创建第三个和第四个对象时，将调用相应的构造函数。相应的构造函数指的是其参数与调用的参数匹配的构造函数。

如果以下面的方式创建一个 Circle 对象，将发生什么情况呢：

```
Circle myCircle = new Circle(1, 2, 3, 4);
```

这将出错，因为它将传递四个值，而 Circle 类没有接受四个值的构造函数，因此上述声明无效。

9.1.3　理解方法的特征标

方法之所以可以重载是由于各个方法都有其独特性。每个方法都有其特征标（signature）。正如前一节介绍的，方法的参数个数能够决定那个方法是合适的。对于被重载方法，还有其他的方式可以区分各个方法。这些差异性最终构成了方法的特征标。

方法的特征标由其参数个数和类型构成。Circle 的构造函数有四个特征标：

- Circle( );
- Circle(int);
- Circle(int, int);
- Circle(int, int, int)。

程序清单 9.1 中的 Area 方法也有 4 个特征标：

- double area( );

- double area(double);
- double area(int, int, double);
- double area(int, int, int, int)。

下述重载方式是合法的：

```
MyFun( int )
MyFun( float )
MyFun( ref int )
MyFun( val int )
```

有些因素不能用作特征标的组成部分。例如返回类型，因为调用方法时，并不使用返回类型。

另外，不能通过一个方法使用某种数据类型，而另一种方法使用一个数组来得到不同的特征标。例如，如果您使用下面两个特征标来重载，将出错：

```
int myMethod( int )
int myMethod( int[])
```

也不能通过关键字 params 来区分特征标。本章后面将介绍如何使用 params。在一个地方声明下面两个方法将出错：

```
void myMethod( string, float)
void myMethod( string, params float[])
```

您可以根据需要重载方法任意多次，只要每一个方法有独特的特征标。

9.2 使用不同数目的参数

前面介绍了如何创建方法、如何给方法传递信息以及传递信息的多种方式——包括按值传递、按引用传递以及传递可用于返回输出的变量。另外，还介绍了使用关键字 return 从方法中返回一个值。所有这一切都要求以有组织的方式使用方法。

当需要给方法传递不同数目的参数时，该如何办呢？例如，假设您需要将一组值累加起来，但并不知道其中到底有多少个值。在这种情况下，可以调用同一个例程多次，或创建一个能接受不同数目参数的例程。请看方法 Console.WriteLine 和 Console.Write，它们都可以接受一个字符串以及不同数目的其他参数，这些参数的数据类型和值可以各不相同。

要接受未知数目的参数，可以使用关键字 params，该关键字用于参数列表中，声明参数列表最后面的值。params 关键字与数组一起使用。

程序清单 9.3 在一个方法中使用了关键字 params，该方法可以接受不同数目的参数，将其累计起来，并将结果作为一个 long 值返回。

程序清单 9.3 使用关键字 params

```
1: // addem.cs - Using a variable number of arguments
2: //---------------------------------------------------
3:
4: using System;
5:
6: public class AddEm
7: {
```

```
 8:     public static long Add( params int[] args )
 9:     {
10:        int ctr = 0;
11:        long Total = 0;
12:
13:        for( ctr = 0; ctr < args.Length; ctr++)
14:        {
15:          Total += args[ctr];
16:        }
17:        return Total;
18:     }
19: }
20:
21: class MyApp
22: {
23:    public static void Main()
24:    {
25:       long Total = 0;
26:
27:       Total = AddEm.Add( 1 );
28:       Console.WriteLine("Total of (1) = {0}", Total);
29:
30:       Total = AddEm.Add( 1, 2 );
31:       Console.WriteLine("Total of (1, 2) = {0}", Total);
32:
33:       Total = AddEm.Add( 1, 2, 3 );
34:       Console.WriteLine("Total of (1, 2, 3) = {0}", Total);
35:
36:       Total = AddEm.Add( 1, 2, 3, 4 );
37:       Console.WriteLine("Total of (1, 2, 3, 4) = {0}", Total);
38:    }
39: }
```

该程序清单的输出如下：

```
Total of (1) = 1
Total of (1, 2) = 3
Total of (1, 2, 3 ) = 6
Total of (1, 2, 3, 4 ) = 10
```

分析：看到该程序清单时，您会说“这可以使用一个简单的整数数组来实现”。如果您这样认为，那么您是完全正确的。对于这种简单的例子，不使用关键字params也能实现，但是，如果不使用关键字params，将无法仅使用第30、33和36行的代码就解决问题。您不能够将值传递给方法，而需要将每个值放在同一个int数组中，然后传递该数组。关键字params让编译器来为您完成这些工作。

仔细查看该程序清单，您将发现它并没有执行什么复杂的工作。第 6～19 行创建了一个名为 AddEm 的类，这个类只有一个名为 Add 静态函数，它接受存储在数组 args 中的不同数目的整数。由于使用了关键字 params，因此可以分别传递整数，而不必将其填充到同一个数组中，然后传递该数组。

AddEm 方法非常简单。第 13～16 行的 for 循环遍历数组 args。记住，该数据是使用传递给方法 AddEm 的整数值创建的。和其他数组一样，您也可以检查该数组的标准属性和方法，这包括使用 args.Length 方法来确定数组的长度。for 循环从数组的开始循环到数组的结尾，并将每个数字加入到累积变量 Total 中。然后在第 17 行将累计值返回给调用方法。

第 21～39 行的 MyApp 类使用 AddEm 方法将所有的数字相加。从中可以看到，调用该方法时，传递的整数个数各不相同。您还可以传递更多的整数来调用该方法，而它仍将正确运行。

注意：这里没有创建AddEm对象，因为Add方法是静态的，因此可以使用类名AddEm来调用它这意味着即使不创建对象，也可以调用该方法。

9.2.1 使用 params 来指定多种数据类型

前一个例子中所有的可变参数都是 int 型的。由于所有的数据类型都是基于同一个类——object，因此可以使用数据类型 object 传递各种不同数据类型的参数。程序清单 9.4 虽然不实用，但却说明了可以传递任何数据类型的值。

程序清单 9.4 grabgae.cs：传递不同数据类型的值

```
 1: // garbage.cs - Using a variable number of arguments
 2: //            of different types
 3: //---------------------------------------------------
 4:
 5: using System;
 6:
 7: public class Garbage
 8: {
 9:    public static void Print( params object[] args )
10:    {
11:       int ctr = 0;
12:
13:       for( ctr = 0; ctr < args.Length; ctr++)
14:       {
15:          Console.WriteLine("Argument {0} is:  {1}", ctr, args[ctr]);
16:       }
17:    }
18: }
19:
20: class MyApp
21: {
22:   public static void Main()
23:   {
24:      long ALong = 123456789098765432lL;
25:      decimal ADec = 1234.56M;
26:      byte Abyte = 42;
27:      string AString = "Cole McCrary";
28:
29:      Console.WriteLine("First call...");
30:      Garbage.Print( 1 );   // pass a simple integer
31:
32:      Console.WriteLine("\nSecond call...");
```

```
33:       Garbage.Print( );    // pass nothing
34:
35:       Console.WriteLine("\nThird call...");
36:       Garbage.Print( ALong, ADec, Abyte, AString );  // Pass lots
37:
38:       Console.WriteLine("\nFourth call...");
39:       Garbage.Print( AString, "is cool", '!' );     // more stuff
40:    }
41: }
```

该程序清单的输出如下：

```
First call...
Argument 0 is:  1

Second call...

Third call...
Argument 0 is:  1234567890987654321
Argument 1 is:  1234.56
Argument 2 is:  42
Argument 3 is:  Cole McCrary

Fourth call...
Argument 0 is:  Cole McCrary
Argument 1 is:  is cool
Argument 2 is:  !
```

分析：第9～17行是Garbage类的Print方法，它被声明为可接受不同数目的对象。任何数据类型都是对象，因此该方法可以接受任何数据类型的值。方法内的代码很容易。从输出中可以知道，第30行首次调用Garbage.Print方法打印的是一个值——1。

在第 32 行，第二次调用该方法时，没有传递任何参数。方法 Garbage.Print 仍将被调用，但该方法将不打印任何东西。在第一次检查 args.Length 的值后，for 语句便结束了。

第三次和第四次对 Garbage.Print 的调用打印其他一些不同的值。通过使用 object 类型，任何数据类型的变量或字面值都可以传递给该方法。

注意：第一周的课程中介绍过，以L结尾的字面值将被视为long值，以M结尾的字面值将被视为decimal（见该程序清单的第24和25行）。

9.2.2　详谈 params

有必要对关键字 params 导致的结果做更详细的解释。当值被传递给方法时，编译器首先查看是否有匹配的方法。如果有，则调用该方法；如果没有，编译器将查看是否有包含参数 params 的方法　如果找到这样的方法，则使用它。编译器将这些值放到一个数组中，并将该数组传递给方法。以程序清单 9.3 中最后一次对 AddEm 类的 Add 方法的调用为例：

```
AddEm.Add(1, 2, 3, 4);
```

编译器将在幕后完成以下工作：

```
int[] x = new int[4];
```

```
int[0] = 1;
int[1] = 2;
int[2] = 3;
int[3] = 4;
AddEm.Add(x);
```

对于程序清单 9.4，编译器将创建并使用一个 object 类型的数组，而不是 int 类型的数组。

警告： 别忘了，数组元素的编号是从0而不是1开始的。

9.2.3 Main 方法和命令参数

您知道，Main 是一个特殊的方法，因为它总是首先被调用。Main 方法也可以接受不同数目的参数，但不需要在 Main 方法中使用 params 关键字。

之所以不需要使用关键字 params，是由于命令行参数将自动加入到一个字符串数组中。这基本上与 params 的功能相同。由于命令行参数已被加入到一个数组中，因此不需要使用关键字 params。

声明 Main 方法时，如果期望传递参数，则标准的做法是使用下面的格式：

```
public static [int | void] Main( string[] args)
```

其中 int 和 void 是可选的。返回整数时使用 int，返回其他数据类型则使用 void。这里的重点是参数列表：一个名为 args 的字符串数组。可将 args 改为任何名称，但几乎所有的 C#程序员都将使用 args。程序清单 9.5 演示了如何使用命令行参数。

程序清单 9.5 command.cs：使用命令行参数

```
1: // command.cs - Checking for command-line arguments
2: //-----------------------------------------------------
3:
4: using System;
5:
6: class CommandLine
7: {
8:    public static void Main(string[] args)
9:    {
10:       int ctr=0;
11:       if (args.Length <= 0 )
12:       {
13:        Console.WriteLine("No Command Line arguments were provided.");
14:        return;
15:       }
16:       else
17:       {
18:         for( ctr = 0; ctr < args.Length; ctr++)
19:         {
20:           Console.WriteLine("Argument {0} is {1}", ctr+1, args[ctr]);
21:         }
22:       }
23:    }
24: }
```

下面是不提供任何参数时，该程序清单的输出：

```
C:\code\Day09>command
No Command Line argument were provided.
```

下面是提供命令行参数时，该程序清单的输出：

```
C:\code\Day09>command xxx 123 456 789.012
Argument 1 is xxx
Argument 2 is 123
Argument 3 is 456
Argument 4 is 789.012
```

分析：该程序清单简单扼要。从第8行开始的Main函数接受命令行参数，其声明中包含参数string[] args，因此将捕获任何命令行参数。第11行检查数组args的数据成员Length。如果为0，则打印一条消息，指出没有提供任何命令行参数；否则执行第18～21行，使用一个for循环打印每一个值。第20行不是从0（而是将计数器加1，从而从1）开始标识各个参数，这样可以避免最终用户有这样的疑问，即为什么第一个参数被称为0。毕竟最终用户不一定是一名C#程序员。

应该	不应该
务必理解方法重载	应尽可能将各种东西设置为私有的，通过属性来提供对私有数据成员的公有访问权限
务必以最常用的方式重载方法	编程时别忽略了命令行参数，可以让程序接受命令行参数，并给操作系统返回一个值

9.3 作用域

新术语：理解被运行阶段环境释放之前，变量的存活时间至关重要。变量的寿命及其可访问性被称为*作用域*。作用域有许多级别，最常用的两个是局部和全局。

全局作用域指的是在整个程序清单中都可见，因此可用。在小范围内可用的变量被称为局部的。

9.3.1 局部作用域

最小的作用域是代码块。代码块可以包含一个简单的循环语句，请看程序清单 9.6 中第 15 行的 x，它的值是多少呢？

程序清单 9.6　scope.cs：在局部变量的作用域之外引用它

```
 1: // scope.cs - Local scope with an error
 2: // *** You will get a compile error ***
 3: //-------------------------------------------------
 4:
 5: using System;
 6:
 7: class Scope
 8: {
 9:   public static voic Main()
10:   {
11:      for( int x; x < 10; x++ )
12:      {
```

```
13:          Console.WriteLine("x is {0}", x);
14:       }
15:       Console.WriteLine("Out of For Loop. x is {0}", x);
16:    }
17: }
```

编译该程序清单时，将出现下面的错误：

```
scope.cs(15,55): error CS0103: The name 'x' does not exist in the class or namespace 'Scope'
```

您可能认为 x 的值应为 10，但实际上在这里引用 x 是错误的。该变量在第 11 行被声明为 for 语句的一部分，for 语句结束后，便在其作用域之外了。离开其作用域，便不能再使用它，因此会出错。

现在来看一下程序清单 9.7。它在 for 语句中包含一个与程序清单 9.6 类似的声明，但在第二个 for 语句中，再次声明了 x 变量，这会导致错误吗？

程序清单 9.7　scope2.cs：声明多个局部变量 x

```
 1: // scope2.cs - Local scope.
 2: //-------------------------------------------
 3:
 4: using System;
 5:
 6: class Scope
 7: {
 8:    public static void Main()
 9:    {
10:       for( int x = 1; x < 5; x++ )
11:       {
12:          Console.WriteLine("x is {0}", x);
13:       }
14:
15:       // Second for statement trying to redeclare x...
16:       for( int x = 1; x < 5; x++ )
17:       {
18:          Console.WriteLine("x is {0}", x);
19:       }
20:    }
21: }
```

该程序清单的输出如下：

```
x is 1
x is 2
x is 3
x is 4
x is 1
x is 2
x is 3
x is 4
```

分析：该程序清单能够运行！其中每个x变量都是其所在代码块（for语句）内的局部变量，因此，所有的x变量都是彼此独立的。

现在来看程序清单 9.8 及其使用的多个 x 变量

程序清单 9.8　scope3.cs：多个 x 变量

```
 1: // scope3.cs - Local scope.
 2: // *** Error if lines are uncommented ***
 3: //-----------------------------------------------
 4:
 5: using System;
 6:
 7: class Scope
 8: {
 9:    static int x = 987;
10:
11:    public static void Main()
12:    {
13:       Console.WriteLine("x is {0}", x);
14:
15: //       for( int x = 1; x < 5; x++ )
16: //       {
17: //          Console.WriteLine("x is {0}", x);
18: //       }
19:       Console.WriteLine("x is {0}", x);
20:    }
21: }
```

该程序清单的输出如下：

```
x is 987
x is 987
```

分析：该程序清单的第15～18行被注释掉了。输入该程序清单时，应原原本本地输入这些注释行，然后编译并执行该程序清单。得到的输出如上所示。正如您预想的，第13行和19行打印的是类中的静态变量x。

第 15～18 行声明了一个局部变量 x，它只能在 for 循环内部使用。根据前一节的介绍，您可能认为，除掉这些代码前面的注释标记后，程序将正常运行——for 循环使用其局部变量 x，而其他方法使用类中的变量 x。但是您错了！编译器如何知道第 17 行使用的不是类中的变量 x 呢？无法知道。除掉第 15～18 行的注释标记，并重新编译该程序清单时，将得到下面的结果：

```
scope3.cs(15,17): error CS0136: A local variable named 'x' cannot be declared in this scope
because it would give a different meaning to 'x', which is already used in a 'parent or current'
scope to denote something else
```

编译器不知道使用哪个 x 变量，局部变量 x 和类的变量 x 之间出现了冲突。有一种方法可以避免这个问题。

9.3.2 区分类变量和局部变量

区分类变量和局部变量的方法之一是总是引用类。本书前面介绍了如何引用，但有必要在这里重温一下。对于程序清单 9.8 中出现的错误，解决方案有两种：一是修改局部变量的名称，一是在第 14 和 19 行明确地指出是类变量。

根据声明变量的方式，明确地引用类变量的方式有两种。如果类变量是标准的、非静态变量，可以使用关键字 this。例如，要访问类的数据成员 x，使用 this.x。

如果数据成员像程序清单 9.8 中那样，是静态的，则使用类名而不是关键字 this 来引用它。有关关键字 this 和如何访问静态数据成员，请参阅第 7 天的课程。

9.3.3 类作用域限定符

对于方法和数据成员，可使用的限定符有两种：private 和 public。前三天的课程中介绍了这两个限定符，本书的后面将介绍其他的限定符。

使用 public 限定符时，数据成员和成员函数在类的外面可以被访问的，前面介绍过很多这样的例子；使用限定符 private 时，数据成员和方法只能在其所在的默认类中被访问。默认情况下，数据成员和方法是私有的。

注意： 如果在类中声明变量时，没有指定是私有的还是公有的，则它将是私有的。

另外，一些语言可以在函数或方法的外面声明变量。这种变量将是全局的，在程序的任何地方都可以使用它。C#不允许在类的外面声明变量。

9.4 不能用于创建对象的类

可能创建一个类，并禁止使用它来创建对象。您可能会问，为何要这样做呢？如果不能创建对象来访问类，那么如何使用它呢？实际上，您已经使用过大量不能用于创建对象的类。例如 Console 类，您在未声明 Console 对象的情况下，便使用了它的 WriteLine 和其他方法。另外，诸如 Math 等类允许您不声明对象便使用它们。

没有对象，如何使用类呢？静态方法和数据成员属于类，而不是各个对象。如果您声明了一个其数据和方法都为静态的类，则使用该类声明的对象将不包含任何值。程序清单 9.9 声明了一个 MyMath 类，其中包含确良多个执行数学运算的方法。

程序清单 9.9 MyMath.cs：执行数学运算的方法

```
 1: // MyMath.cs - Static members.
 2: //---------- ----------------------------------------
 3:
 4: using System;
 5:
 6: public class MyMath
 7: {
 8:    public static long Add( params int[] args )
 9:    {
10:       int ctr = 0;
11:       long Answer = 0;
12:
13:       for( ctr = 0; ctr < args.Length; ctr++)
```

```
14:       {
15:         Answer += args[ctr];
16:       }
17:       return Answer;
18:     }
19:
20:     public static long Subtract( int arg1, int arg2 )
21:     {
22:       long Answer = 0;
23:       Answer = arg1 - arg2;
24:       return Answer;
25:     }
26: }
27:
28: class MyApp
29: {
30:   public static void Main()
31:   {
32:     long Result = 0;
33:
34:     Result = MyMath.Add( 1, 2, 3 );
35:     Console.WriteLine("Add result is {0}", Result);
36:
37:     Result = MyMath.Subtract( 5, 2 );
38:     Console.WriteLine("Subtract result is {0}", Result);
39:   }
40: }
```

该程序清单的输出如下：

```
Add result is 6
Subtract result is 3
```

分析：第6～26行的MyMath类声明了两个方法：Subtract和Add，它们对整数执行加法和减法运算，并将结果返回。这些方法可以更复杂些，这些工作留给读者去完成。

虽然没有理由去创建一个 MyMath 对象，但您可以创建这样的对象，不过确实可以禁止创建某种对象。

9.4.1　私有构造函数

要禁止使用某种类来创建对象，可以创建一个私有的构造函数。正如前面介绍的，带关键字 private 的方法只能在其所属的类中调用。这意味着不能在类的外面调用私有构造函数。由于构造函数是在创建对象时被调用的，因此给构造函数加上限定符 private 实际上可以禁止创建对象。程序清单 9.10 包含一个构造函数为私有的 MyMath 类。

程序清单 9.10　MyMath2.cs：构造函数为私有的 MyMath 类

```
1: // MyMath2.cs - Private constructor
2: //---------------------------------------------------
3:
```

```
 4: using System;
 5:
 6: public class MyMath
 7: {
 8:    public static long Add( params int[] args )
 9:    {
10:       int ctr = 0;
11:       long Answer = 0;
12:
13:       for( ctr = 0; ctr < args.Length; ctr++)
14:       {
15:          Answer += args[ctr];
16:       }
17:       return Answer;
18:    }
19:
20:    public static long Subtract( int arg1, int arg2 )
21:    {
22:       long Answer = 0;
23:       Answer = arg1 - arg2;
24:       return Answer;
25:    }
26:
27:    private MyMath()
28:    {
29:       // nothing to do here since this will never get called!
30:    }
31: }
32:
33: class MyApp
34: {
35:   public static void Main()
36:    {
37:       long Result = 0;
38:
39:       // MyMath var = new MyMath();
40:
41:       Result = MyMath.Add( 1, 2, 3 );
42:       Console.WriteLine("Add result is {0}", Result);
43:
44:       Result = MyMath.Subtract( 5, 2 );
45:       Console.WriteLine("Subtract result is {0}", Result);
46:    }
47: }
```

该程序清单的输出如下：

```
Add result is 6
Subtract result is 3
```

分析：第27～30行是类的构造函数。如果删除第39行的注释标记，并重新编译该程序清单，将发生下面的错误：

```
MyMath3.cs(39,20): error CS0122: 'MyMath.MyMath( )' is inaccessible due to its protection
level
```

无法创建对象。限定符 private 禁止您创建对象。但这并不是什么问题，因为您可以访问公有的静态类成员。

9.5　再谈名称空间

名称空间可用于帮助组织类和其他类型。您已经使用过多个由框架提供的名称空间，其中包括 System 名称空间，该名称空间包含大量系统方法和类，其中的 Console 类包含读写信息的例程。

名称空间可以包含其他名称空间、类、结构、枚举、接口和代表。其中名称空间、类、结构和枚举，您已经熟悉了，本书后面将介绍接口和代表。

9.5.1　给名称空间命名

名称空间的名称可以是任何合法的标识符，这意味着其名称必须由标准字符和下划线组成，另外还可以包含句点。和其他标识符一样，应该给名称空间取一个描述性的名称。

9.5.2　声明名称空间

要创建名称空间，可以使用关键字 namespace 后面跟标识该名称空间的名称。然后使用花括号便该名称空间包含的类型括起。

正如前面指出的，名称空间可以包含其他名称空间。名称空间的格式化方式与前面介绍的相同。程序清单 9.11 声明了一个名称空间。

程序清单 9.11　namesp.cs：声明名称空间

```
1:  // namesp.cs - Declaring namespaces
2:  //---------------------------------------------------------
3:
4:  using System;
5:
6:  namespace Consts
7:  {
8:     public class PI
9:     {
10:       public static double value = 3.14159;
11:       private PI() {}  // private constructor
12:    }
13:    public class three
14:    {
15:       public static int value = 3;
16:       private three() {} // private constructor
17:    }
18: }
19:
20: namespace MyMath
```

```
21: {
22:  public class Routine
23:  {
24:    public static long Add( params int[] args )
25:    {
26:      int ctr = 0;
27:      long Answer = 0;
28:
29:      for( ctr = 0; ctr < args.Length; ctr++)
30:      {
31:        Answer += args[ctr];
32:      }
33:      return Answer;
34:    }
35:
36:    public static long Subtract( int arg1, int arg2 )
37:    {
38:      long Answer = 0;
39:      Answer = arg1 - arg2;
40:      return Answer;
41:    }
42:  }
43: }
44:
45: class MyApp
46: {
47:   public static void Main()
48:   {
49:     long Result = 0;
50:
51:     Result = MyMath.Routine.Add( 1, 2, 3 );
52:     Console.WriteLine("Add result is {0}", Result);
53:
54:     Result = MyMath.Routine.Subtract( 5, 2 );
55:     Console.WriteLine("Subtract result is {0}", Result);
56:
57:     Console.WriteLine("\nThe value of PI is {0}", Consts.PI.value );
58:     Console.WriteLine("The value of three is {0}",Consts.three.value);
59:   }
60: }
```

该程序清单的输出如下：

```
Add result is 6
Subtract result is 3

The value of PI is 4.14159
The value of three is 3
```

分析：该程序清单在前一个程序清单的基础上做了修改，另外，还声明了其他几个不实用的类，但它确实有助于说明名称空间的概念。

该程序清单声明了两个名称空间，第 6 行是第一个。名称空间 Consts 包含两个类：PI 和 three，第 57 和 58 行分别使用了这两个类。在这些代码行中，必须指定名称空间、类和数据成员，从其他名称空间中（如第 57 和 58 行）访问这些类时，如果省略 Consts，将出错：

```
namesbad.cs(20,1): error CS1529: A using clause must precede all other namespace elements
```

但可以使用关键字 using，来避免这种错误。在程序清单的开始位置添加下面的语句，将可以消除这种错误：

```
using Consts;
```

注意：每个文件都包含了一个全局名称空间，即使不明确声明。该全局名称空间的所有东西在文件中指定的每个名称空间中都是可用的。

9.5.3　using 和名称空间

C#提供了关键字 using，它使得使用名称空间更为容易，并提供了两项功能。首先，可以使用 using 给名称空间取一个别名；其次，使用 using 后，不必使用全限定名称，因此访问名称空间中的类型将更为容易。

9.5.3.1　避免使用全限定名称

您已经知道如何使用 using 来避免使用全限定名称。通过包含下面的代码行：

```
using System;
```

在使用名称空间 System 中的类和类型时，不用再包含名称空间名称 System。这样，您可以使用 Console.WriteLine，而不用包含名称空间名称 System。对于程序清单 9.11，可以在第 5 行加入如下所示的代码：

```
using Consts;
```

这样，您便可以使用 PI.value 和 three.value，而不必使用名称空间 Consts 进行限定。

警告：必须将 using 语句放在其他代码元素之前，这意味着最好将它放在程序清单的开始位置。如果试图将其放在其他位置，将出错。

9.5.3.2　使用 using 提供别名

也可以使用关键字 using 给名称空间提供别名，这让您能够给名称空间（甚至名称空间中的类）取一个不同的名称。别名可以是任何合法的标识符，取别名的格式如下：

```
using aliasname = namespaceOrClassName;
```

其中 aliasname 是要使用的别名，namespaceOrClassName 是名称空间或类的名称。例如下面的代码行：

```
using doit = System.Console;
```

如果在程序清单中包含上述代码行，则可以在需要使用 System.Console 的地方使用 doit。这样，要将一行文本写到屏幕上，可以使用下面的代码：

```
doit.WriteLine( "blah blah blah");
```

程序清单 9.12 使用 System.Console 的别名演示了“Hello World”程序

程序清单 9.12　useit.cs：使用 using 提供别名

```
1: // useit.cs
2: //-------------------------------------------------------
3:
4: using doit = System.Console;
5:
6: class MyApp
7: {
8:    public static void Main()
9:    {
10:      doit.WriteLine("Hello World!");
11:   }
12: }
```

该程序清单的输出如下：

```
Hello World!
```

分析：该程序清单简单易懂。第4行使用using给System.Console创建了一个别名doit。然后在第10行使用该别名来打印一条消息。

应该	不应该
务必理解作用域	应尽可能将数据成员设置为私有的
应使用关键字 using 来简化对名称空间成员的访问	别忘了，默认情况下，数据成员是私有的
应使用名称空间来组织您的类	

9.6 总　结

今天的课程对前几天所学的知识进行了扩展，介绍了如何重载方法，以便能够接受不同数据和类型的参数。这可以通过创建具有独特特征标的重载方法来实现。除了重载常规方法外，还介绍了如何重载类的构造函数。

另外，今天还更详细地介绍了类成员的作用域。关键字 private 使得成员只在其所属的类中可用；而关键字 public 则使得可以在类所属的成员外访问成员。另外，还可以创建只存活于代码块内部的变量。this 关键字用于标识类的特定实例的数据成员。

今天还介绍了名称空间，其中包含如何创建自己的名称空间。在讨论名称空间时，还介绍了关键字 using，该关键字让您引用名称空间的成员时，不用包含全限定名称；另外它还可用于给名称空间或类取别名。

9.7 问与答

问：可以将 Main 方法声明为私有的吗？

答：可以，但这样将无法访问该方法。要运行程序，Main 方法必须是公有的，如果不能在 Main 方法所在的类的外面访问它，将无法运行程序。

问：今天的课程简要的介绍了作用域。如果没有明确地给变量指定值，则其默认值是什么？

答：很多类型的变量一开始并没有被赋值，这包括没有被初始化的结构实例变量、输出参数和局部变量；而很多类型的变量开始被赋了值，这包括静态变量、对象实例变量、被初始化的结构实例变量、数组元素、用作方法参数的值变量和引用。虽然这些变量一开始被赋了值，但对于所有的变量，都应该设置其初始值。

问：为什么不将所有的东西都声明为公有的以简化它们？

答：面向对象语言的优点之一是能够将数据和函数封装到可以被看作是一个黑盒子的类中。将成员设置为私有的，使得内部结构的修改不影响使用类的程序成为可能。

9.8 作　业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

9.8.1 小测验

1. 重载函数是封装、继承、多态还是重用的方式之一？
2. 成员函数可以被重载多少次？
3. 下面哪些成员可被重载：

a. 数据成员；

b. 成员方法；

c. 构造函数；

d. 析构函数。

4. 哪个关键字用于使方法能够接受不同数目的参数？
5. 如何使方法可以接受不同数目、不同（甚至未知）数据类型的参数？
6. 要从命令行接受不同数目的参数，需要包含哪个关键字？
7. 类成员的默认作用域是什么？
8. 限定符 public 和 private 之间的差别何在？
9. 如何禁止使用类实例化对象？
10. using 关键字有哪两种用途？

9.8.2 练习

1. 为名为 abc 的公有函数编写方法头，该函数接受两个 short 参数，返回值类型为 byte。
2. 编写一行接受命令行参数的代码。
3. 如果有一个名为 aClass 的类，可以加入什么代码来禁止该类被用于实例化对象。
4. 下面的程序有问题吗？在编辑器中输入该程序，并编译它。如果有问题，是什么问题？

```
1: using doit = System.Console.WriteLine;
2:
3: class MyApp
4: {
5:    public static void Main()
```

```
6:    {
7:       doit("Hello World!");
8:    }
9: }
```

5. 创建一个名称空间，它包含一个类和另一个名称空间，被包含的名称空间也包含一个类。然后创建一个使用这两个类的应用程序类。

6. 创建一个包含被重载方法的程序，该重载方法有以下特征标，这些特征标都合法吗？

```
public myFunc( )
public myFunc( int )
public myFunc( float )
public myFunc( ref int )
public myFunc( ref flaot )
```

第 10 天课程
处理异常

如果每个人编写的代码都是完美无缺的，每位用户都输入正确的信息，而所有的计算机都不出现错误，则当今的大量程序将无用武之地。实际情况是，计算机、用户和程序员都不可避免地会犯错误。因此，编写程序时，必须预防意外情况。另外，编写程序时，还需要处理另一种不同的问题——异常。今天的课程包含以下内容：

- 异常处理的概念；
- 关键字 try 和 catch；
- 通过关键字 finally；
- 一些常见的异常及其引发原因；
- 将异常传递给其他例程；
- 定义自己的异常；
- 引发和再引发异常。

10.1　异常处理的概念

创建程序时，需要考虑各种可能出现的问题。创建那些使用来自文件、用户、服务或程序的另一部分信息的程序时，应该对信息进行检查，以确保您收到的或使用的信息是您预料之中的。来看一下第 2 天的练习 2 中创建的程序 List1，它要求您在运行程序时指定一个文件名。如果没有指定文件名将出现什么情况？如果指定的文件不存在将出现什么情况？如果指定的是一个机器码文件将出现什么情况呢？如果指定了多个文件名又将出现什么情况呢？

您当然可以不管这些问题，但用户将放弃使用您的程序。这种错误的后果各不相同。编写程序时，您应该做出这样的决定，即哪些问题足够严重，需要考虑，哪些问题不需要。优秀的程序员总会对那些可能发生的异常和意外情况进行预防。

10.1.1　通过逻辑代码预防错误

通过在代码中加入简单的编程逻辑，便可以处理大量的问题。如果一种简单的编程逻辑便可以防止一种错误，则应该加入这种逻辑。例如，您可以检查值的长度、是否有代码行参数或核实某个数字是否在合法的范围之内。使用第一周学习的编程结构，可以很容易地实现这种检查。

请考虑下列情况，如何在代码中处理这些情况呢？

- 用户试图打开一个不存在的文件；
- 给数组指定过多的元素；
- 程序中的代码将一个字符串赋给整型变量；
- 例程试图使用包含空值的引用变量。

可以编写代码来避免这些问题，但万一遗漏了一个，将出现什么情况呢？

10.1.2 导致异常的原因

如果不有计划地捕获问题，将发生异常。异常是未被捕获的编程错误，这部包括逻辑错误，逻辑错误指的是结果不正确，而不是由于编码方面的问题引起的。当发生意外的错误时，运行阶段环境将终止，并引发异常。程序清单 10.1 包含一个将引发异常的错误。运行该程序清单，看看当您访问包含五个元素的数组的第六个元素（不存在）时，将出现什么情况。

程序清单 10.1 Error.cs：导致异常

```
1: // error.cs
2: // A program that throws an exception
3: //=========================================
4: using System;
5:
6: class MyError
7: {
8:    public static void Main()
9:    {
10:      int [] myArray = new int[5];
11:
12:      for ( int ctr = 0; ctr < 10; ctr++ )
13:      {
14:         myArray[ctr] = ctr;
15:      }
16:   }
17: }
```

分析：编译该程序清单时不会出现错误。该程序清单并不实用，因为它没有提供实际的输出或执行任何有价值的操作。第10行创建了一个名为myArray的、包含五个元素的Int数组。第12～15行的for循环将计数器的值赋给该数组的每一个元素。0被赋给了myArray[0]，1被赋给了myArray[1]，等等。

当 ctr 的值等于 5 时，将出现什么情况呢？第 12 行的条件语句使得这要 ctr 的值小于 10，就不断得将其递增。但当 ctr 的值等于 5 后，出现了问题。在第 14 行 myArray[5]是非法的——该数组的最后一个元素是 myArray[4]。运行阶段环境知道该数组的索引值不能为 5，因此它引发错误，指出出现了异常情况。当您运行该程序时，可能出现一个窗口，指出异常错误，如图 10.1 所示。另外，运行阶段环境还会显示如下的晦涩文本：

```
Exception occurred: System.IndexOutOfRangeException: A exception of type
System.IndexOutOfRangeException was throw at MyError.Main( )
```

上述文本实际上是由您的程序正在使用的一个底层对象——数组生成的。

程序清单 10.1 的结果不合适，您不希望用户看到这样的信息。为使程序对用户是友好的，您需要以一种更为友好的方式来处理这种异常错误。另外，该程序在异常发生时突然终止，您希望在

异常发生时，仍能够控制程序。

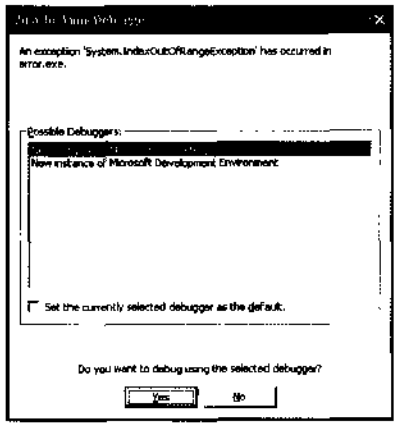


图 10.1　运行阶段环境显示的异常错误

提示：对程序清单10.1进行修改，在给每个元素赋值后，打印它的值。方法是在第14行的后面加入下面的代码：

```
Console.WriteLine("myArray[{0}] equals {1}", ctr, myArray[ctr]);
```

（运行该程序清单，您将发现程序将在打印 myArray[5]之前终止。）

10.2　异常处理

异常处理指的是处理诸如程序清单 10.1 中出现的运行阶段错误。可以在程序中加入代码，提供更为明确的结果，而不是弹出窗口、终止程序并显示晦涩的消息。为此，需要使用关键字 try 和 catch。

10.2.1　使用 try 和 catch

关键字try和catch对于异常处理至关重要。try命令让您能够给代码块加上包装，帮助发送（route）任何可能发生的异常。

关键字 catch 让您能够捕获 try 命令发送的异常。通过使用 catch，您能够执行代码并控制出现的问题，而不是让程序终止。程序清单 10.2 演示了命令 try 和 catch 的一种基本用途。

程序清单 10.2　tryit.cs：使用 try–catch

```
 1: // tryit.cs
 2: // A program that throws an exception
 3: //=============================================
 4: using System;
 5:
 6: class MyAppClass
 7: {
 8:    public static void Main()
 9:    {
10:       int [] myArray = new int[5];
```

```
11:
12:       try
13:       {
14:           for ( int ctr = 0; ctr < 10; ctr++ )  // Array only has 5 elements!
15:           {
16:               myArray[ctr] = ctr;
17:           }
18:       }
19:
20:       catch
21:       {
22:         Console.WriteLine("The exception was caught!");
23:       }
24:
25:    Console.WriteLine("At end of class");
25:    }
27: }
```

该程序清单的输出如下：

```
The exception was caught!
At end of class
```

分析：该程序清单与10.1类似，只是使用try和catch添加了基本的异常处理而已。该程序清单的主要代码被封装到一个try语句中。该try语句是从第12行开始的，它使用花括号（第13行和18行）将要监视其异常的代码块——即操纵数组的代码——括起。

try 语句后面是从第 20 行开始的 catch 语句，该语句由第 21 行和 23 行的花括号之间的语句组成。如果在执行 try 语句中的代码时发生异常，程序流程将立刻跳到 catch 语句，执行 catch 语句中的代码（而不是像程序清单 10.1 那样显示一条晦涩的消息，并终止程序），并继续执行程序。在程序清单 10.2 中，catch 语句打印一条消息并继续执行程序，因此第 25 行对 WriteLine 方法的调用仍将被执行。

10.2.2 捕获异常信息

在程序清单 10.2 中，catch 语句将捕获 try 语句的代码执行时可能发生的任何异常。除了捕获引发的异常外，还可以在 catch 中包含参数来确定引发的是哪种异常。catch 语句的格式如下：

```
catch( System.Exception e) {}
```

catch 语句能够将异常作为一个参数接受。在这个例子中，异常是一个名为 e 的变量。可以使用更具描述性的名称，但就这个例子而言，名称 e 是管用的。

从上面可以知道，变量 e 的类型为 System.Exception，这是一个全限定名称，表明 Exception 类型是在名称空间 System 中定义的。如果使用 using 语句将 System 名称空间包含进来，则可以将上述 catch 语句简化为：

```
catch( Exception e) { }
```

Exception 变量 e 中包含关于发生的特定异常的描述性信息。程序清单 10.3 在程序清单 10.2 的基础上进行了修改，包含一条将任何异常都作为参数进行接受的 catch 语句。其中修改过的代码行

以粗体显示。

程序清单 10.3　tryit2.cs：捕获异常信息

```
1:  // tryit2.cs
2:  // A program that throws an exception
3:  //=============================================
4:  using System;
5:
6:  class MyAppClass
7:  {
8:    public static void Main()
9:    {
10:     int [] myArray = new int[5];
11:
12:     try
13:     {
14:         for ( int ctr = 0; ctr < 10; ctr++ )  // Array only has 5 elements!
15:         {
16:             myArray[ctr] = ctr;
17:         }
18:     }
19:
20:     catch( Exception e)
21:     {
22:       Console.WriteLine("The following exception was caught:\n{0}", e);
23:     }
24:
25:     Console.WriteLine("At end of class");
26:   }
27: }
```

该程序清单的输出如下：

```
The following exception was caught:
System.IndexOutOfRangeException: An exception of type System.IndexOutOfRangeException was
thrown at MyAppClass.Main()
At end of class
```

分析：该程序清单并没有对异常做太多的处理，但确实提供了不少的信息。第22行使用方法WriteLine打印变量e的值，即显示关于异常的信息。从输出中可以知道，e的值表明引发的异常是IndexOutOfRangeException，发生在MyAppClass（应用程序类）的Main()方法中。

该程序清单将捕获 try 语句中发生的所有异常，打印的错误取决于异常的类型。实际上，可以在程序中加入代码来处理特定的异常。

注意：同样，异常也被捕获，而程序继续执行。使用catch语句，可以防止自动显示怪异的错误消息。另外，程序也不会在异常发生时终止。

10.2.3 给 try 语句提供多个 catch 语句

程序清单 10.2 中的 catch 语句过于全面，它将捕获 try 语句中的代码发生的所有错误。可以使用更为具体的 catch 语句，事实上可以编写专门捕获特定异常的 catch 语句。程序清单 10.4 包含一条专门捕获您熟悉的异常 IndexOutOfRangeException 的 catch 语句。

程序清单 10.4 catchIndex.cs：捕获特定的异常

```
1:  // catchIndex.cs
2:  // A program that throws an exception
3:  //=======================================
4:  using System;
5:
6:  class MyAppClass
7:  {
8:    public static void Main()
9:    {
10:     int [] myArray = new int[5];
11:
12:     try
13:     {
14:        for ( int ctr = 0; ctr < 10; ctr++ )  // Array only has 5 elements!
15:        {
16:           myArray[ctr] = ctr;
17:        }
18:     }
19:
20:     catch (IndexOutOfRangeException e)
21:     {
22:        Console.WriteLine("You were very goofy trying to use a bad array index!!", e);
23:     }
24:
25:     catch (Exception e)
26:     {
27:       Console.WriteLine("Exception caught: {0}", e);
28:     }
29:
30:     Console.WriteLine("\nDone with the catch statements. Done with program.");
31:   }
32: }
```

该程序清单的输出如下：

```
You were very goofy trying to use a bad array index!!

Done with the catch statements. Done with program.
```

分析：该程序清单使用的数组和try命令与前几个程序清单相同，但第20～23行包含一些新内容。第20行的catch语句包含一个IndexOutOfRangeException类型的参数，而不是一个通用异常参数。和通

用的Exception类型一样，IndexOutOfRangeException类型也位于名称空间System中。顾名思义，该异常类型是专门用于索引超出范围的情况的，因此catch语句将只捕获这种异常。

为应付可能发生的其他异常，第 25～28 行包含了另一个 catch 语句。该 catch 语句使用的是通用的 Exception 类型的参数，因此将捕获可能发生的其他任何异常。将该程序清单的第 16 行改为如下所示：

```
16:          myArray[ctr] = 100/ctr;   // division by zero....
```

重新编译并运行该程序清单，将出现如下所示的输出：

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
   at MyAppClass.Main( )

Done with the catch statements. Done with program.
```

修改后的第 16 行将导致另一种错误。第一次执行第 14 行的 for 循环时，ctr 等于 0，因此第 16 行执行的运算为 100 除以 0（ctr）。被 0 除是非法的，因此这将导致无穷大，所有引发一个异常。这种异常不是索引超出范围，因此第 20 行的 catch 语句将被忽略，因为该异常与 IndexOutOfRangeException 不匹配。第 25 行的 catch 语句将捕获任何异常，因此被执行。第 27 行打印文本“Exception Caught:”和变量 e 中的异常描述。从输出可以知道，引发的异常是 DivideByZeroException。

10.2.4　理解异常的处理顺序

在程序清单 10.4 中，两个 catch 语句的顺序至关重要。总是在前面捕获较具体的异常，在后面捕获较通用的异常。对于程序清单 10.4，如果改变两个 catch 语句的顺序：

```
catch (Exception e)
{
    Console.WriteLine("Exception caught: {0}", e);
}

catch (IndexOutOfRangeException e)
{
     Console.WriteLine("You were very goofy trying to use a bad array index!!", e);
}
```

然后重新编译该程序清单，将出错。这是因为通用的 catch(Exception e)将捕获所有的异常，因此其他的 catch 语句将不会被执行。

10.3　使用 finally 添加最后执行的操作

有时候，不管 try 语句中的代码是否成功执行，都需要执行一个代码块。为此，C#提供了关键字 finally。finally 语句块中的代码总是会执行。程序清单 10.5 演示了如何使用该关键字。

程序清单 10.5　final.cs：使用关键字 finally

```
1: // final.cs
2: // A program that throws an exception
```

```
 3: //---------------------------------------------
 4: using System;
 5:
 6: class MyAppClass
 7: {
 8:    public static void Main()
 9:    {
10:       int [] myArray = new int[5];
11:
12:       try
13:       {
14:          for ( int ctr = 0; ctr < 10; ctr++ )  // Array only has 5 elements!
15:          {
16:             myArray[ctr] = ctr;
17:          }
18:       }
19:
20: //     catch
21: //     {
22: //        Console.WriteLine("Exception caught");
23: //     }
24:
25:       finally
26:       {
27:          Console.WriteLine("Done with exception handling");
28:       }
29:
30:       Console.WriteLine("End of Program");
31:    }
32: }
```

该程序清单的输出如下：

```
Exception occurred: System.IndexOutOfRangeException: An exception of type
System.IndexOutOfRangeException was thrown.
   at MyAppClass.Main()
Done with exception handling
```

分析：该程序清单与前面的几个相同，主要区别在于第25～28行，这里加入了一个finally子句，它打印一条消息。注意，虽然catch子句（第20～23行被注释掉了）没有捕获任何异常，但在程序结束前，finally仍将被执行。但第30行的WriteLine方法没有被执行。

删除第 20～23 行的注释标记，重新编译并执行该程序，将得到如下所示的结果：

```
Exception caught.
Done with exception handling
End of Program
```

使用 catch 语句后，finally 中的代码仍将执行。如果将第 14 行修改为如下所示：

```
14:          for ( int ctr = 0; ctr < 5; ctr++ )
```

然后重新编译并执行该程序，则输出如下：

```
Done with exception handling
End of Program
```

对第 14 行修改后，消除了导致异常的问题，这意味着该程序清单运行时不会出现问题。从输出可以知道，finally 代码块仍将被执行。由此可见，不管情况如何，finally 语句块都将被执行。

下面介绍一个更强大的使用异常处理的例子。程序清单 10.6 是一个更为实用的程序。

程序清单 10.6　ListFile.cs：使用异常处理

```
 1: // ListFile.cs - program to print a listing to the console
 2: //---------------------------------------------------------
 3:
 4: using System;
 5: using System.IO;
 6:
 7: class ListFile
 8: {
 9:   public static void Main(string[] args)
10:   {
11:     try
12:     {
13:
14:       int ctr=0;
15:       if (args.Length <= 0 )
16:       {
17:         Console.WriteLine("Format: ListFile filename");
18:         return;
19:       }
20:       else
21:       {
22:         FileStream fstr = new FileStream(args[0], FileMode.Open);
23:         try
24:         {
25:           StreamReader t = new StreamReader(fstr);
26:           string line;
27:           while ((line = t.ReadLine()) != null)
28:           {
29:             ctr++;
30:             Console.WriteLine("{0}: {1}", ctr, line);
31:           }
32:         }
33:         catch( Exception e )
34:         {
35:           Console.WriteLine("Exception during read/write: {0}\n", e);
36:         }
37:         finally
38:         {
```

```
39:             fstr.Close();
40:           }
41:       }
42:     }
43:
44:     catch (System.IO.FileNotFoundException)
45:     {
46:       Console.WriteLine ("ListFile could not find the file {0}", args[0]);
47:     }
48:     catch (Exception e)
49:     {
50:       Console.WriteLine("Exception: {0}\n\n", e);
51:     }
52:   }
53: }
```

如果运行该程序清单时，不提供任何参数，则输出如下：

```
Format: ListFile filename
```

如果运行该程序时，提供参数 ListFile.cs，则输出的将是该程序清单，且包含行号：

```
 1: // ListFile.cs - program to print a listing to the console
 2: //---------------------------------------------------------
 3:
 4: using System;
 5: using System.IO;
 6:
 7: class ListFile
 8: {
 9:   public static void Main(string[] args)
10:   {
11:     try
12:     {
13:
14:       int ctr=0;
15:       if (args.Length <= 0 )
16:       {
17:         Console.WriteLine("Format: ListFile filename");
18:         return;
19:       }
20:       else
21:       {
22:         FileStream fstr = new FileStream(args[0], FileMode.Open);
23:         try
24:         {
25:           StreamReader t = new StreamReader(fstr);
26:           string line;
27:           while ((line = t.ReadLine()) != null)
28:           {
29:             ctr++;
```

```
30:                Console.WriteLine("{0}:  {1}", ctr, line);
31:              }
32:           }
33:           catch( Exception e )
34:           {
35:             Console.WriteLine("Exception during read/write: {0}\n", e);
36:           }
37:           finally
38:           {
39:             fstr.Close();
40:           }
41:        }
42:     }
43:
44:     catch (System.IO.FileNotFoundException)
45:     {
46:       Console.WriteLine ("ListFile could not find the file {0}", args[0]);
47:     }
48:     catch (Exception e)
49:     {
50:       Console.WriteLine("Exception: {0}\n\n", e);
51:     }
52:   }
53: }
```

也可以以其他文件名为参数，如果该文件存在，则输出将与此类似。如果参数指示的文件不存在，则屏幕上将显示如下所示的消息（其中 xxx 为指定的文件名）：

```
List could not find the file xxx
```

注意，该程序运行时，运行阶段环境不会给用户显示晦涩的异常消息，而是提供有用的信息，指出发生的情况。这是通过结合使用编程逻辑和异常处理实现的。

该程序清单将前面介绍的知识组合在一起。第 4 和 5 行不但包含了名称空间 System，还包含了 System 中的名称空间 IO。IO 名称空间包含了用于发送和接收（输入和输出）信息的例程。

从第 7 行开始是应用程序类 ListFile，该类包含一个 Main( )方法，程序从这里开始执行。第 9 行的 Main( )方法接收一个名为 args 的字符串数组作为参数，args 中的值来自您运行程序时包含的命令行参数。

从第 11 行开始，是与今天重点介绍的内容相关的代码。这一行声明了一个 try 代码块，该代码块包含第 11～42 行的代码，它包含大量的代码，其中包括还有另一个 try 命令。如果其中的任何代码导致异常发生，而这种异常又没有被处理，则 try 语句将失败，程序流程将进入相应的 catch 代码块。注意，仅当 try 代码块中发生未被处理的异常时，程序流程才会进入其 catch 语句。

与该最优先的 try 语句对应的 catch 代码块有两个。第一个位于第 44～47 行，它捕获特定异常 FileNotFoundException。出于清晰方面的考虑，使用的是全限定异常名，但可以省略其中的名称空间，因为第 5 行已经将名称空间 System.IO 包含进来了。FileNotFoundException 异常在您试图打开一个不存在的文件时发生。在这个例子中，如果文件不存在，第 46 行将打印一条简单的消息，指出文件找不到。

虽然该程序处理了 FileNotFoundException 异常，但还是加入了第 48～51 行，以防出现其他意外的异常。这样程序可以得体地结束，而不是依赖于运行阶段环境。

下面详细地探讨 try 语句中的代码，以便您能更深入地理解该程序的功能。第 14 行创建了一个简单的计数器 ctr，用于给程序清单加上行号。

第 15 行包含了确保用户运行程序时提供一个文件名的编程逻辑。如果没有提供文件名，则结束程序。它使用 if 语句来检查字符串数组 args 的 Length 属性，如果数组长度不大于 0，则说明用户没有提供命令行参数。用户必须提供至少一个命令行参数，如果没有提供任何参数，则显示一条描述性消息，并使用 return 语句结束程序。

如果用户输入了命令行参数——args.Length 大于 0，则执行第 20～41 行的 else 语句。第 22 行创建了一个名为 fstr 的新对象，其类型为 FileStream，它有一个接受两个参数的构造函数。第一个是文件名。您传递的是用户输入的用户名，它位于 args[0]中；第二个是一个关于执行何种操作的指示器，您传递的是 FileMode.Open，它指示 FileStream 对象打开一个文件，以便能够读取其内容。您使用创建的 FileStream 对象 fstr 来引用该文件。

如果第 22 行失败，并引发一个异常，将跳到第 44 行的 catch 语句。该行的 catch 语句与 try 语句最近。

从第 23 行开始是一个新的 try 代码块，其对应的 catch 语句位于第 33 行。第 25 行创建一个名为 t 的 StreamReader 变量，该变量与第 22 行打开的文件 fstr 相关。文件被看作是一个流向程序的字符流，变量 t 用于读取该字符流。

第 26 行声明了一个名为 line 的字符变量，用于存储流向程序的一系列字符。第 27 行演示了如何使用变量 line

第 27 行完成的工作很多，有必要对其进行详细地剖析。首先使用 t 读取一行字符。StreamReader 类有一个名为 ReadLine 的方法，它读取一行字符。一行字符指的是换行符之前的所有字符。由于 t 与 fstr 相关，而后者与用户输入的文件相关，因此 ReadLine 方法将返回用户指定的文件中的下一行字符。然后，该行字符被赋给字符串变量 fstr，读取一行字符，并将其赋给变量 line 后，将 line 与 null 进行比较，如果返回的字符串为空，则说明到了文件的结尾处或发生了读取错误。无论是哪种情况，当遇到 null 后，都没有必要继续处理该文件。

如果读取并放入到 line 的字符串不为空，则执行 while 语句中的代码块。这里是将行计数器 ctr 加 1，并打印一行文本，其中包含行号和 line 中的内容。这一过程将一直重复下去，直到 line 的值为 null 位置。

如果从文件中读取一行时遇到问题，很可能会引发异常。第 33～36 行将不获任何可能发生的异常，并显示描述性文本和异常消息。该 catch 语句防止运行阶段环境接管控制权，并通过提供额外的关于什么地方出了问题的信息，以帮助您和用户。

第 37～40 行包含一个 finally 语句，它对应于第 23 行的 try 语句。该 finally 语句不做任何工作，而只是关闭第 22 行打开的文件。由于第 22 行已成功执行——如果失败，则它将引发异常，从而跳到第 44 行的 catch 语句——因此，在程序结束前，需要关闭该文件。无论第 24～42 行是否出现异常，在程序结束前，都应该将文件关闭，而 finally 语句确保 Close 方法被调用。

正如您在该程序清单中看到的，try-catch-finally 语句可以嵌套。另外，还可用于使程序对用户更友好。

注意：本书中程序清单的行号就是使用一个与ListFile极其类似的程序加上的。

10.4　常见的异常

大量的异常被定义为.NET 框架类、前面已经介绍过其中的几个。表 10.1 列出了 System 名称空间中许多常见的异常类。

表 10.1　　System 名称空间中常用的异常类

异常类名称	描　述
MemberAccessException	访问错误：类型成员（如方法）不能被访问
ArgumentException	参数错误：方法的参数无效
ArgumentNullException	参数为空：给方法传递一个不可接受的空参数
ArithmeticException	数学计算错误：由于数学运算导致的异常，其覆盖面比 DivideBeZeroException 和 OverflowException 更大
ArrayTypeMismatchException	数组类型不匹配：试图将不兼容类型的数据存储到数组中时引发
DivideByZeroException	被零除：试图除以零时引发
FormatException	格式不正确：参数的格式不正确
IndexOutOfRangeException	索引超出范围：使用的索引小于 0 或比最后一个数组元素的索引还大
InvalidCastException	非法强制转换：显式转换失败时引发
MulticastNotSupportedException	不支持的组播：组合两个非空代表（将在第 14 天的课程中介绍）失败时引发
NotFiniteNumberException	无限大的值：数字不合法
NotSupportedException	方法不被执行：调用的方法在类中没有实现
NullReferenceException	空引用：引用空引用对象时引发
OutOfMemoryException	内存不足：无法为新语句分配内存时引发
OverflowException	溢出：使用 checked 关键字后，数学运算导致所赋的值过大或过小时引发
StackOverflowException	栈溢出：栈中的命令过多时引发
TypeInitializationException	错误的初始化类型：静态构造函数有问题时引发

注意：表10.1中的名称是假设您使用using语句包含了System名称空间；否则需要使用全限定名称System.ExceptionName，其中ExceptionName是表中使用的名称。

10.5　定义自己的异常类

除了框架中定义好的异常外，您还可以创建自己的异常。在 C#中，应该引发异常，而不是传回大量不同的错误码。因此总是应该在代码中处理异常，以防出现异常，这很重要。虽然这会增加程序中的代码，但可以使程序对用户更友好。

创建自己的异常类后，您可能想让它发生。为此，需要引发（throw）异常。要引发自己的异常，需要使用关键字 throw。

您可以引发预定义的异常，也可以引发自己的异常。预定义的异常指的是已经在名称空间中定义好的异常。例如，您可以引发表 10.1 中的所有异常。为此，需要使用关键字 throw，其格式如下：

```
throw (exception);
```

如果该异常不存在，还需要使用关键字 new 来创建它。例如，程序清单 10.7 引发一个新的 DivideByZeroException 异常。虽然该程序清单毫无作用，但它确实以最基本的方式说明了关键字 throw。

注意： 使用关键字throw时，括号是可有可无的，下面的两行代码等效：

```
throw (exception);
throw exception;
```

程序清单 10.7 zero.cs：引发异常

```
 1: // zero.cs
 2: // Throwing a predefined exception.
 3: // This listing gives a runtime exception error!
 4: //=============================================
 5: using System;
 6:
 7: class SimpleApp
 8: {
 9:    public static void Main()
10:    {
11:       Console.WriteLine("Before Exception...");
12:       throw( new DivideByZeroException() );
13:       Console.WriteLine("After Exception...");
14:    }
15: }
```

该程序清单的输出如下：

```
Before Exception..

Exception occurred: System.DivideBeyZeroException: Attempted to divide by zero.
at SimpleApp.Main( )
```

分析： 该程序清单只是打印一条消息，并在第12行引发DivideByZeroException异常而已。程序执行时，出现一个运行错误，指出引发了异常。它简单，但也不实用。

编译并运行该程序时，将出现运行错误；第 13 行永远不会被执行，因为 throw 命令将终止程序。记住，throw 命令将立刻离开当前的例程。可以删除第 13 行（因为它不会被执行），以免编译器提出警告。这里之所以加入这行代码，只是为了强调异常对程序流程的影响。

注意： 可以将该程序清单中的DivideByZeroException换成表10.1列出的任何异常，而输出中将出现相应的信息。

10.6　引发自己的异常

也可以创建并引发自己的异常，这种异常也更有价值。要创建自己的异常，必须首先声明它。声明的格式如下：

```
class ExceptionName: Exception { }
```

其中 ExceptionName 是要声明的异常的名称。这行代码指出您的异常是一个类，并与一个已有的类 Exception 相关。在关于继承的明天的课程中，将更详细地介绍这种关系。

提示：异常的名称以Exception结尾，从表10.1可以知道，这与预定义异常一致。

创建自己的异常只需一行代码，创建后便可以捕获它。程序清单 10.8 演示了如何创建并引发自己的异常。

程序清单 10.8　throwit.cs：创建并引发自己的异常

```
// throwit.cs
// Throwing your own error.
//=========================================
using System;

class MyThreeException : Exception {}

class MyAppClass
{
   public static void Main()
   {
      int result;

      try
      {
         result = MyMath.AddEm( 1, 2 );
         Console.WriteLine( "Result of AddEm(1, 2) is {0}", result);

         result = MyMath.AddEm( 3, 4 );
         Console.WriteLine( "Result of AddEm(3, 4) is {0}", result);
      }

      catch (MyThreeException)
      {
         Console.WriteLine("Ack!  We don't like adding threes.");
      }

      catch (Exception e)
      {
        Console.WriteLine("Exception caught: {0}", e);
      }

      Console.WriteLine("\nAt end of program");
   }
```

```
35: }
36:
37: class MyMath
38: {
39:    static public int AddEm(int x, int y)
40:    {
41:       if(x == 3 || y == 3)
42:          throw( new MyThreeException() );
43:
44:       return( x + y );
45:    }
46: }
```

该程序清单的输出如下：

```
Result of AddEm(1, 2) is 3
Ack! We don't like adding threes.

At end of program
```

分析：该程序清单演示了如何创建自己的异常MyThreeException。该异常是在第6行定义的，其格式与前面介绍的相同。这让您能够引发一个基本的异常。

介绍 MyAppClass 之前，我们先来看看位于第 37～46 行的第二个类。这个类名为 MyMath，只包含一个简单的静态方法 AddEm。AddEm 方法将两个数相加，并将结果返回。第 41 行使用 if 语句检查传递给 AddEm 的值是否至少有一个为 3，如果是，则引发第 6 行声明的 MyThreeException 异常。

第 8～34 行是 MyAppClass 类中的 Main 方法，它调用 AddEm 方法。这种调用是在 try 语句中进行的，这样如果引发了异常，便可以做出反应。第 16 行是对 AddEm 的第一次调用，传递的值是 1 和 2。这些值不会引发异常，因此程序将继续执行。第 19 行再次调用方法 AddEm，这次传递的第一个值为 3，它将导致 AddEm 方法引发异常 MyThreeException。第 23 行包含一个捕获 MyThreeException 的 catch 语句，因此该语句将捕获并处理这种异常。

如果不捕获异常，运行阶段环境将显示一条异常消息。如果将程序清单 10.8 的第 23～26 行注释掉，然后重新编译并运行该程序，将得到如下所示的输出：

```
Result of AddEm(1, 2) is 3
Exception caught: MyThreeException: An exception of type MyThreeException was thrown.
at MyAppClass.Main( )

At end of program
```

该消息与其他异常完全相同。您也可以给处理异常的 catch 语句传递一个参数，其中包含用于通用系统消息的信息。例如，如果将第 23～26 行修改为如下所示：

```
23:       catch (MyThreeException e )
24:       {
25:          Console.WriteLine("Ack! We don't like adding threes. \n {0}" e);
26:       }
```

则得到的结果如下所示：

```
Result of AddEm(1, 2) is 3
Ack! We don't like adding threes.
MyThreeException: An exception of type MyThreeException was thrown.at MyAppClass.Main( )

At end of program
```

新创建的异常的运行方式与已有的异常完全相同。

提示：程序清单10.8创建了一个基本异常。为使之更为完整，应该给该异常加上三个构造函数。在明天学习继承后，您对这些重载方法的细节将更为清楚。就现在而言，您只需知道加入下述包含三个构造函数的代码后，异常将更为完整：

```
class MyThreeException : Exception
{
   public MyThreeException()
   {
   }

   public MyThreeException( string e ) : base (e)
   {
   }

   public MyThreeException( string e, Exception inner ) :
       base ( e, inner )
   {
   }
}
```

您可以将异常名MyThreeException替换为自己的异常名。

10.6.1　重新引发异常

既然可以引发自己的异常和系统异常，便可以重新引发已有的异常。为何要这样做呢？什么时候这样做呢？

您知道，可以捕获异常并通过执行自己的代码来做出反应。如果上述操作是在一个类中进行的，而这个类又被另一个类所调用，则您可能希望将问题告诉调用类，但之前可能自己做一些处理。

来看一个基于前面显示文件内容的程序的例子，您可以创建打开文件，计算其包含的字符数，并将结果返回给调用程序的类。如果在打开文件并计算其中的字符时发生错误，将会引发异常。您捕获该异常，将字符数设置为 0，并将其返回给调用程序。这样，调用程序将只知道返回的字节数为 0。

一种更佳的反应方式是，将字符数设置为 0，并再次引发异常，让调用程序来处理。这样调用程序将知道发生的情况。

要重新引发异常，需要在 catch 语句中包含一个异常类型的参数。下面的代码是能够重新引发异常，让调用方法对其进行处理的 catch 语句：

```
catch ( Exception e)
{
   //My personal exception logic here
   throw( e ); //e is the argument received by this catch
```

```
}
```

注意：当您开始编写更复杂的应用程序时，可能想进一步了解异常处理技术。今天，您学习了最重要的异常处理特性，此外还有很多其他的特性，但这种主题已超出了本书的范围。

10.7 checked 语句和 unchecked 语句

有两个 C#关键字决定了将过大或过小的值赋给变量是否会引发异常，它们是 checked 和 unchecked。如果代码是 checked，则将过大或过小的值赋给变量以引发异常。如果代码 unchecked，则将所赋的值进行裁剪，以便将其存储到变量中。程序清单 10.9 演示了这两个关键字的用法。

程序清单 10.9 checkit.cs：使用关键字 checked

```
1: // checkit.cs
2: //=========================================
3:
4: using System;
5:
6: class MyAppClass
7: {
8:    public static void Main()
9:    {
10:      int result;
11:      const int topval = 2147483647;
12:
13:      for( long ctr = topval - 5L;  ctr < (topval+10L); ctr++ )
14:      {
15:        checked
16:        {
17:           result = (int) ctr;
18:           Console.WriteLine("{0} assigned from {1}", result, ctr);
19:        }
20:      }
21:   }
22: }
23:
```

该程序清单的输出如下：

```
2147483642 assigned from 2147483642
2147483643 assigned from 2147483643
2147483644 assigned from 2147483644
2147483645 assigned from 2147483645
2147483646 assigned from 2147483646
2147483647 assigned from 2147483647

Exception occurred: System.OverflowException: An exception of type System.OverflowException
was thrown.
```

```
   at MyAppClass.Main()
```

分析：第11行声明了一个名为topval的固定变量，其值为int变量能够存储的最大值2147483647。第13行的for循环的计数器上限为最大值加10。该上限被放在一个long变量中，因此万事大吉。但17行显式地将ctr的值赋给int变量result。执行该程序清单时，将出错，因为第16～19行的代码将被检查，而这些代码试图将一个比result变量能够存储的最大值还大的值赋给它。

注意：如果将第13行的+10删除，并重新编译该程序清单，则它将正常运行。这是因为没有错误。仅当所赋的值比topval大时，溢出错误才会发生。

现在修改该程序清单，以使用关键字 unchecked。将第 15 行修改为如下所示：

```
15:     unchecked
```

重新编译并执行该程序清单。该程序清单将通过编译，但输出结果出乎意料，如下所示：

```
2147483642 assigned from 2147483642
2147483643 assigned from 2147483643
2147483644 assigned from 2147483644
2147483645 assigned from 2147483645
2147483646 assigned from 2147483646
2147483647 assigned from 2147483647
-2147483648 assigned from 2147483648
-2147483647 assigned from 2147483649
-2147483646 assigned from 2147483650
-2147483645 assigned from 2147483651
-2147483644 assigned from 2147483652
-2147483643 assigned from 2147483653
-2147483642 assigned from 2147483654
-2147483641 assigned from 2147483655
-2147483640 assigned from 2147483656
```

这次没有出现异常，这是因为代码将不被检查（unchecked）。但结果却不是您想要的。

10.7.1　checked 和 unchecked 的格式

在程序清单 10.9，checked 和 unchecked 被用作语句。它们的格式如下：

```
[un]checked {//statements}
```

也可以将它们用作运算符，其格式如下：

```
[un]checked (expression)
```

其中要检查（或不检查）的表达式被放在括号中。

警告：决不要认为默认为checked或unchecked。通常，默认为checked，但有些因素将改变这种默认情况　对于命令行编译器，可以通过加入/checked选项，强行检查；如果您使用的是集成开发工具，则编译选项中应该包含检查项。可以在命令行中加入/checked-选项，以不进行检查.

10.8　总　结

今天介绍了异常处理。try 语句用于检查发生的异常，在发生异常时，可以通过 catch 语句以控

制权更大的方式来处理错误。可以为 try 语句提供多个 catch 语句，这样便可以定制不同异常的处理方式，并使用 Exception 来捕获所有的基本异常。

可以创建在异常处理代码（try 和 catch）语句执行后被执行的代码块，无论是否引发了异常，这种代码块都将被执行。这种代码块使用关键字 finally 进行标记。

最后讨论了两个与处理溢出错误相关的关键字：checked 和 unchecked。将过大值赋给变量时将发生溢出错误。可以使用关键字 checked 检查代码或表达式的是否溢出，也可以使用关键字 unchecked 忽略溢出错误。

10.9 问与答

问：看起来单独使用 catch 的功能最为强大，为何不在使用 catch 时，不带任何参数，并在其中自己完成所有的逻辑呢？

答：虽然单独使用 catch 的功能最为强大，但这样将无法知道关于引发异常的信息。因此，使用 catch(Exception e)更佳。这让您能够获得关于引发异常的信息。如果不使用这些信息，则可以将其传递给其他类，这样这些类就可以选择是否使用这些信息。

问：所有的异常将被平等对待吗？

答：不会。实际上，异常分为两类：系统异常和应用程序异常。前者将终止应用程序，而后者不会。今天介绍的大部分是更为常见的异常——应用程序异常。更详细的关于异常以及这两类异常之间的差别的信息，请查看.NET 框架文档或 C#文档。

问：前面指出过，关于异常处理还有大量的内容，我需要学习这些知识吗？

答：今天介绍的关于异常处理的知识对于编写程序而言足够了。通过进一步学习异常处理方面的知识，将能够更好地操纵错误和消息。另外，您还知道如何将一个异常嵌入到另一个异常中，等等。并不一定要知道这些高级概念，但知道这些概念将使您成为一名更优秀、更高级的 C#程序员。

10.10 作 业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

10.10.1 小测验

1. 哪些关键字用于处理异常？
2. 下面的情况哪些应通过异常进行处理，哪些应通过常规代码进行处理？

a. 用户输入的值不在指定的范围之内。

b. 无法正确地读取文件。

c. 传递给方法的参数包含一个无效的值。

d. 传递给方法的参数的类型非法。

3. 什么会导致异常？
4. 异常是在什么时候发生的？

a. 编写程序时。

b. 编译时。

c. 运行时。

d. 最终用户发出请求时。

5. 哪个关键字用于对异常做出反应?

6. finally 代码块何时执行?

7. 单个 try 语句可以有多少个 catch 语句与之对应?

8. catch 语句的次序有关系吗? 为什么?

9. 大多数预定义的常用异常都是在哪个名称空间中定义的?

10. throw 命令的功能是什么?

10.10.2　练习

1. 可以使用什么代码来检查下述代码行是否引发了异常?

```
GradePercentage = MyValue / Total;
```

2. 下面的程序有问题，错误的原因是什么?

```
int zero = 0;
try
{
     int result = 1000 / zero;
}

catch (Exception e)
{
    Console.WriteLine("Exception caught: {0}", e);
}
catch (DivideByZeroException e)
{
     Console.WriteLine("This is my error message. ", e);
}
finally
{
     Console.WriteLine("Can't get here");
}
```

3. 编写创建自己的异常类的代码，该异常类包含今天的补充材料中建议的三个被重载的构造函数。将该异常命名为 NegativeValueException。

4. 在一个完整的程序清单中使用练习 3 中创建的 NegativeValueException 类。请在程序清单 10.8 的基础上创建该程序清单，并创建一个名为 SubtractEm 类，它在减法运算的结果为负值时，引发异常 NegativeValueException。

第 11 天课程

继　　承

面向对象语言最重要的功能之一是扩展已有的类，这可以通过继承来实现。今天介绍以下内容：

- 基类；
- 通过继承来扩展基类；
- 通过继承来扩充类的功能；
- 保护数据，但允许派生类共享；
- 对类进行密封；
- 使用关键字 is 和 as 将对象作为不同的类型进行操纵。

11.1　继承的基本知识

我们都从父母那里继承了一些特征，如眼睛的颜色、头发的颜色和质地（texture）等。除了这些特征外，我们还有扩展的特征。就想我们从父母那里继承并扩展特征一样，类也可以从其他类派生而来。父母的一些特征将被后代的特征覆盖，而一些特征不会。

继承让我们能够在已有类的基础上创建新类。新类可以使用原有类的所有特性，可以覆盖已有的特性、扩展已有的特性或添加自己的特性。

同一个类可以派生多个类，但每个类都只能从一个类派生而来。例如，您有一个名为 control 的基类，它可以派生出许多不同的控件类型。图 11.1 帮助说明了这种一对多的关系。从图的左边往右看，便可以看到这种一对多的关系；但从右边往左看，可以发现每个类都是从一个类派生而来的。

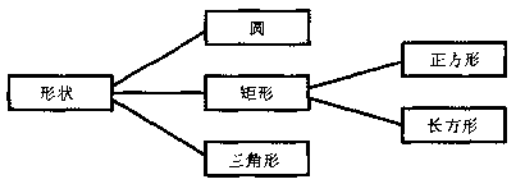


图 11.1　继承关系

讨论继承时，常常使用一些基本术语：

- 基类：原来的类；
- 父类：同基类的含义相同；

- 派生类：通过继承基类创建的新类；
- 子类：含义与派生类相同。
- 单继承：派生类是使用一个（且只能是一个）基类创建的。C#只支持单继承。图 11.1 说明的便是单继承。
- 多重继承：派生类是使用两个或更多的基类创建的。C#不支持多重继承。

上面列出的并不是与继承相关的所有重要术语，今天的课程还将介绍许多其他术语。

单继承和多重继承

与 C++不同，C#不支持多重继承。多重继承指的是新的派生类是使用多个基类创建的。例如，可以使用姓名类和地址类派生出一个业务联系类。后者将包含前两个类的特征。多重继承将导致许多问题，并增加复杂程度。通过使用接口（将在第 13 天的课程中介绍），可以实现多重继承的大部分功能。

11.1.1　简单继承

理解继承的最佳方式是使用它。要继承，首先需要有基类。程序清单 11.1 包含一个今天将用来说明继承的类，同时将使用该程序清单的一部分。

程序清单 11.1　基类及其用法

```
1:  // inherit01.cs
2:  // A relatively simple class to use as a starting point
3:  //--------------------------------------------------
4:  using System;
5:  using System.Text;
6:
7:  class Person
8:  {
9:    private string firstName;
10:   private string middleName;
11:   private string lastName ;
12:   private int    age;
13:
14:   // ToDo: Add properties to access the data members
15:
16:   public Person()
17:   {
18:   }
19:
20:   public Person(string fn, string ln)
21:   {
22:     firstName = fn;
23:     lastName = ln;
24:   }
25:
26:   public Person(string fn, string mn, string ln)
27:   {
28:     firstName = fn;
```

```
29:       middleName = mn;
30:       lastName = ln;
31:    }
32:
33:    public Person(string fn, string mn, string ln, int a)
34:    {
35:       firstName = fn;
36:       middleName = mn;
37:       lastName = ln;
38:       age = a;
39:    }
40:
41:    public void displayAge()
42:    {
43:       Console.WriteLine("Age {0}", age);
44:    }
45:
46:    public void displayFullName()
47:    {
48:       StringBuilder FullName = new StringBuilder();
49:
50:       FullName.Append(firstName);
51:       FullName.Append(" ");
52:       if( middleName[0] != "" )
53:       {
54:          FullName.Append(middleName[0]);
55:          FullName.Append(". ");
56:       }
57:       FullName.Append(lastName);
58:
59:       Console.WriteLine(FullName);
60:    }
61: }
62:
63: // NameApp class. Illustrates the use of the Person class
64: class NameApp
65: {
66:    public static void Main()
67:    {
68:       Person me = new Person("Bradley", "Lee", "Jones");
69:       Person myWife = new Person("Melissa", "Anne", "Jones", 21);
70:
71:       me.displayFullName();
72:       me.displayAge();
73:
74:       myWife.displayFullName();
75:       myWife.displayAge();
76:    }
77: }
```

该程序清单的输出如下：

```
Bradley L. Jones
Age 0
Melissa A. Jones
Age 21
```

分析：用于继承的类Person是在第7～61行定义的。虽然仅仅为了说明继承，这个类远不用这么复杂，但这也使得它更为实用。第9～12行包含类的四个数据成员，它们存储关于人的信息。每个数据成员的存取限定符都是private，private使得对数据成员的访问仅限于类的内部。要在这个类的外部修改这些值，需要加入属性。

注意：为缩短该程序清单的长度，省略了属性。您应该加入属性，也可以将存取限定符改为public，以便能够在外部访问这些数据成员，但建议不要这样做。最好是封装数据成员，并使用属性。

第 16－39 行包含四个构造函数，这让用户可以以多种方式创建 Person 对象。另外，这个类还包含两个成员方法，第 41～44 行是一个简单的 displayAge 方法，它显示人的年龄。第 46－60 行是一个较为复杂的方法 dispalyFullName。

DisplayFullName 方法使用了一个以前没有介绍过的新类 StringBuilder，它位于名称空间 System 中。具体地说，是位于名称空间 System.Text 中。第 5 行将名称空间 System.Text 包含了进来，以便能够方便地使用 StringBuilder 类。StringBuilder 类创建一个字符串，这种字符串可以以普通字符串不能的方式进行操纵。例如，字符串的长度不能修改。与普通字符串相同，修改 StringBuilder 对象或追加字符更容易。

第 48 行创建了一个 StringBuilder 对象 FullName，用来保存格式化后的人的全名。第 50 行将姓 firstName 追加到新创建的字符串 FullName 中。由于之前 FullName 为空，因此第 50 行基本上是将姓复制到 FullName 中。第 51 行在姓的后面加上一个空格。第 52～56 行将中名的大写首字母（而不是整个中名）加入到 FullName 中。第 52 行使用 if 语句确保中名不为 “”（空字符串）。如果有中名，则第 54 行将其首写字母追加到 FullName 中，然后第 55 行追加一个句点和空格。最后第 57 行追加名。

最后一行将全名显示到屏幕上。您可能需要将该方法的名称改为 getFullName，并返回格式化后的姓名，这样调用程序可以以其他方式使用全名。

程序清单的余下部分是 NameApp 类，之所以提供这个类，是为了让您看到使用 Person 类的情况。第 68 和 69 行声明了两个名为 my 和 myWife 的 Person 对象。第 71～75 行调用了这些对象的方法。

11.1.2 使用继承

虽然程序清单 11.1 中的代码很多，但您应该完全理解了它。除了 StringBuilder 类之外，这里并没有什么新东西。但程序清单 11.2 中包含了几个新特性。例如，使用下面的格式从基类继承：

```
class derived_class : base_class
```

冒号将新的 derived_class 类与原有的 base_class 类分开，表示继承。

程序清单 11.2 inherit02.cs：基本继承

```
1: // inherit02.cs
2: // Basic inheritance.
```

```
3: //=======================================
4: using System;
5: using System.Text;
6:
7: class Person
8: {
9:   protected string firstName;
10:  protected string middleName;
11:  protected string lastName;
12:  private int  age;
13:
14:  //ToDo: Add properties to access data members
15:
16:  public Person()
17:  {
18:  }
19:
20:  public Person(string fn, string ln)
21:  {
22:    firstName = fn;
23:    lastName = ln;
24:  }
25:
26:  public Person(string fn, string mn, string ln)
27:  {
28:    firstName = fn;
29:    middleName = mn;
30:    lastName = ln;
31:  }
32:
33:  public Person(string fn, string mn, string ln, int a)
34:  {
35:    firstName = fn;
36:    middleName = mn;
37:    lastName = ln;
38:    age = a;
39:  }
40:
41:  public void displayAge()
42:  {
43:    Console.WriteLine("Age {0}", age);
44:  }
45:
46:  public void displayFullName()
47:  {
48:     StringBuilder FullName = new StringBuilder();
49:
50:     FullName.Append(firstName);
51:     FullName.Append(" ");
52:     if( middleName != "" )
```

```
53:       {
54:          FullName.Append(middleName[0]);
55:          FullName.Append(". ");
56:       }
57:       FullName.Append(lastName);
58:
59:       Console.WriteLine(FullName);
60:    }
61: }
62:
63: class Employee : Person
64: {
65:    private ushort hYear;
66:
67:    public ushort hireYear
68:    {
69:      get { return(hYear); }
70:      set { hYear = value; }
71:    }
72:
73:    public Employee() : base()
74:    {
75:    }
76:
77:    public Employee( string fn, string ln ) : base( fn, ln)
78:    {
79:    }
80:
81:    public Employee(string fn, string mn, string ln, int a) :
82:             base(fn, mn, ln, a)
83:    {
84:    }
85:
86:    public Employee(string fn, string ln, ushort hy) : base(fn, ln)
87:    {
88:      hireYear = hy;
89:    }
90:
91:    public new void displayFullName()
92:    {
93:      Console.WriteLine("Employee: {0} {1} {2}",
94:                        firstName, middleName, lastName);
95:    }
96: }
97:
98: class NameApp
99: {
100:    public static void Main()
101:    {
102:       Person myWife = new Person("Melissa", "Anne", "Jones", 21);
```

```
103:        Employee me = new Employee("Bradley", "L.", "Jones", 23);
104:        Employee you = new Employee("Kyle", "Rinni", 2000);
105:
106:        myWife.displayFullName();
107:        myWife.displayAge();
108:
109:        me.displayFullName();
110:        Console.WriteLine("Year hired: {0}", me.hireYear);
111:        me.displayAge();
112:
113:        you.displayFullName();
114:        Console.WriteLine("Year hired of him: {0}", you.hireYear);
115:        you.displayAge();
116:    }
117: }
```

提示：程序清单变得很长。如果不想输入所有这些代码，可以从Sam出版社的网站（www.sampublishing.com）或作者的网站（www.teachyourselfcsharp.com）下载。

该程序清单的输出如下：

```
Melissa A. Jones
Age 21
Employee: Bradley L. Jones
Year hired: 0
Age 23
Employee: Kyle  Rinni
Year hired of him: 2000
Age 0
```

分析：该程序清单演示了最简单的继承方式。在今天课程的后面您将知道，当您开始编写更为复杂的程序时，一些问题便出现了。现在，您应重点理解该程序清单完成的工作。

第 4～61 行包含的代码几乎与程序清单 11.1 相同。但对类做了修改，您注意到了吗？第 9～11 行的存取限定符不同了，不是 private，而是 protected。由于派生类是一个位于基类外面的新类，因此不能访问基类的私有变量，基类的私有变量只能在基类的内面访问。限定符 protected 也有一定的限定性，但同时让派生类能够访问数据成员。Person 类中修改的地方就是这些。今天课程的后面将介绍将类作为基类时，需要修改的一些地方。

第 63～96 行是一个新的派生类 Employee。第 63 行使用了前面提到过的冒号。从基类 Person 派生出了一个新的 Employee 类，后者包含前者的所有功能。

第 65 行给 Employee 添加了一个名为 hYear 的数据成员，用于保存雇员被雇佣时的年份。这是一个私有变量，通过使用第 67～71 行声明的属性来访问。

第 73～89 行是 Employee 类的构造函数，其中也使用了冒号，指示方式不同而已。请看第 77 行，这一行声明了 Employee 的一个构造函数，它接受两个字符串参数：fn 和 ln。

冒号后面是关键字 base，可以以这种方式使用关键字 base 来调用基类的构造函数。第 77 行调用基类的构造函数 Person，并将传递给构造函数 Employee 的两个参数（fn 和 ln）传递给它。当基类的构造函数执行完毕后，将执行构造函数 Employee 中的代码（第 78 和 79 行）。这里没有额外的代

码。在第 86～95 行的构造函数中，则使用属性 hireYear 来设置 hYear 的值。

第 91～95 行是 Employee 类的 displayFullName 方法，声明中使用了关键字 new。由于这个方法的名称与基类的一个成员方法相同，同时使用了关键字 new，因此该方法将覆盖基类中同名的方法。通过 Employee 对象调用 displayFullName 方法时，都将执行第 91～95 行的代码，而不是第 46～60 行的代码。

程序清单的最后声明了一个 NameApp 类，以演示如何使用派生类。第 102～104 行声明了三个对象，其他的代码则调用这些对象的方法。第 106 行调用 myWife 对象的 displayFullName 方法，由于 myWife 被声明为一个 Person 对象，因此显示的是 Person 类中该方法的输出，其中名使用的是缩写。第 107 行调用了 myWife 对象的 displayAge 方法。

第 109 行调用了 me 对象的 displayFullName 方法。由于在第 103 行，me 被声明为一个 Employee 对象，因此调用的将是 Employee 类中的该方法。第 111 行调用了 me 对象的 dispalyAge 方法。由于 Employee 类没有 displayAge 方法，因此程序将自动检查基类是否有这样的方法，然后使用基类的 displayAge 方法。

第 110 行使用 Employee 类的属性显示 hireYear。如果试图使用 myWife 对象的 hireYear，将出现什么情况呢？如果在第 107 行后面加上下面的代码行，将出现什么情况呢？

```
Console.WriteLine("Year hired: {0}", myWife.hireYear);
```

这将出错。不能通过基类访问派生类中的方法，而 Person 类又没有 hireYear 数据成员，因此这行代码非法。

11.1.3 在派生类的方法中使用基类的方法

关键字 base 也可用于直接调用基类的方法。例如，将第 91～95 行改为如下所示：

```
public new void displayFullName( )
{
   Console.Write("Employee: ");
   Base.displayFullName( );
}
```

这修改了派生类中新的 displayFullName 方法。现在，它不是自己来完成所有的工作，而是先显示一部分文本，然后调用基类的 displayFullName 方法。这让您能够扩展基类的功能，而不用重新编写所有的代码。

11.2 多 态

至此您学会了继承的简单应用。您知道，可以通过添加数据成员和方法来扩展基类，还可以使用关键字 base 来调用基类的方法。您还知道使用 new 关键字定义同名的方法，来覆盖基类的方法。

虽然这些都管用，但由于某些原因，还可能需要完成其他的工作。面向对象语言的重要概念之一是多态。如果继承得正确，则类将具有多态方面的优点。具体地说，您将能够创建一系列以层次方式排列的类，并以几乎相同的方式处理它们。

来看 Employee 和 Person 类。Employee 是 Person，但又不仅仅是 Person，它拥有 Person 的一切，还有其他东西。程序清单 11.1 最大限度地压缩了 Person 和 Employee 的例子，请注意该程序清单的 NameApp 中的声明。

程序清单 11.3 inherit03.cs：给 Employee 和 Person 变量赋值

```
// inherit03.cs
//=======================================
using System;

class Person
{
   protected string firstName;
   protected string lastName;

   public Person()
   {
   }

   public Person(string fn, string ln)
   {
      firstName = fn;
      lastName = ln;
   }

   public void displayFullName()
   {
      Console.WriteLine("{0} {1}", firstName, lastName);
   }
}

class Employee : Person
{
   public ushort hireYear;

   public Employee() : base()
   {
   }

   public Employee( string fn, string ln ) : base( fn, ln)
   {
   }

   public Employee(string fn, string ln, ushort hy) : base(fn, ln)
   {
      hireYear = hy;
   }

   public new void displayFullName()
   {
      Console.WriteLine("Employee: {0} {1}", firstName, lastName);
   }
}

```

```
49: class NameApp
50: {
51:    public static void Main()
52:    {
53:       Employee me = new Employee("Bradley", "Jones", 1983);
54:
55:       Person Brad = me;
56:
57:       me.displayFullName();
58:       Console.WriteLine("Year hired: {0}", me.hireYear);
59:
60:       Brad.displayFullName();
61:    }
62: }
```

该程序清单的输出如下：

```
Employee: Bradley Jones
Year hired: 1983
Bradley Jones
```

分析：该程序清单的重点是第53和55行。第53行创建了一个名为me的Employee对象，并给它赋值。第55行创建了一个名为Brad的Person对象，并将其设置为等于Employee对象me，即将Employee对象赋给了Person对象。这是如何完成的呢？前面讲过，Employee是Person，而又不仅仅是Person。Employee包含Person的所有功能。从更广泛的角度说，基类的所有东西都是派生类的一部分。

可以像使用其他 Person 对象一样使用 Brad 对象。第 60 行调用了 displayFullName 方法，而全名确实是使用 Person 类的 displayFullName 方法来显示的。

由于已经将一个 Employee 对象赋给了 Person 对象 Brad，那么可以调用 Employee 类的方法或使用其数据成员吗？例如，可以使用 Brad.hireYear 吗？这很容易进行测试，方法是在第 60 行后面添加下述代码：

```
Console.WriteLine("Year hired: ", Brad.hireYear);
```

您可能认为这将显示 1983，但是您错了！Brad 是一个 Person 对象，而 Person 类没有数据成员 hireYear，因此上述代码将导致编译错误：

```
Inherit03.cs(61,45): error CS0117: 'person' does not contain a definition for 'hireYear'
```

虽然 Employee 包含 Person 的一切，但 Person 并不包含 Employee 的一切，即使将一个 Employee 赋给它。从更广泛的角度说，派生类包含基类的一切，但基类却没有包含派生类的一切。

这种多态指的是什么呢？简单地说，您可以对多种对象类型调用同一个方法，而这种调用有效。在这个例子中，您对 Person 和 Employee 调用 displayFullName 方法，虽然这两个对象被赋给了相同的值，但调用的却是各自所属类的 displayFullName 方法。您不用指定要调用的是哪个类的方法。

11.3 虚拟方法

在面向对象编程中，使用到派生对象的基类引用的情况很常见。在前一个例子中，Brad 被声明

为一个 Person 变量，但却被赋给了一个 Employee 对象。在这种情况下，调用 displayFullName 方法时，实际调用的将是哪个类中相应的方法呢？基类的，该变量被赋给了一个 Employeee 对象。在大多数情况下，您希望调用被赋给的对象所属类中的方法。

新术语： *在C#中，这是通过虚拟方法实现的。虚拟方法让您能够调用实际被赋给的对象所属类中的方法，而不是基类的方法。*

要在基类中将方法声明为虚拟的，可以在该方法的定义中使用关键字 virtual。如果这种方法在派生类中被覆盖（overridden），则在运行阶段，调用的将是变量实际所属类的方法，而不是其被声明的类的方法。这意味着可以使用基类来指向多个派生类，而调用相应的方法。

要覆盖虚拟方法，必须在派生类中指出，这是通过在声明新方法时使用关键字 override 实现的程序清单 11.4 对 11.3 做了修改，请注意输出中不同的地方。

注意： *为减少代码，该程序清单删除了一些构造函数。对于这个例子而言，这没有影响*

程序清单 11.4　inherit04.cs：使用虚拟方法

```
1:  // inherit04.cs - Virtual Methods
2:  //=============================================
3:  using System;
4:
5:  class Person
6:  {
7:     protected string firstName;
8:     protected string lastName;
9:
10:    public Person()
11:    {
12:    }
13:
14:    public Person(string fn, string ln)
15:    {
16:       firstName = fn;
17:       lastName = ln;
18:    }
19:
20:    public virtual void displayFullName()
21:    {
22:       Console.WriteLine("{0} {1}", firstName, lastName);
23:    }
24: }
25:
26: class Employee : Person
27: {
28:    public ushort hireYear;
29:
30:    public Employee() : base()
31:    {
32:    }
33:
34:    public Employee(string fn, string ln, ushort hy) : base(fn, ln)
```

```
35:    {
36:      hireYear = hy;
37:    }
38:
39:    public override void displayFullName()
40:    {
41:      Console.WriteLine("Employee: {0} {1}", firstName, lastName);
42:    }
43: }
44:
45: // A new class derived from Person...
46: class Contractor : Person
47: {
48:    public string company;
49:
50:    public Contractor() : base()
51:    {
52:    }
53:
54:    public Contractor(string fn, string ln, string c) : base(fn, ln)
55:    {
56:      company = c;
57:    }
58:
59:    public override void displayFullName()
60:    {
61:      Console.WriteLine("Contractor: {0} {1}", firstName, lastName);
62:    }
63: }
64:
65: class NameApp
66: {
67:    public static void Main()
68:    {
69:
70:       Person Brad = new Person("Bradley", "Jones");
71:       Person me   = new Employee("Bradley", "Jones", 1983);
72:       Person Greg = new Contractor("Hill", "Batfield", "Data Diggers");
73:
74:       Brad.displayFullName();
75:       me.displayFullName();
76:       Greg.displayFullName();
77:    }
78: }
```

该程序清单的输出如下：

```
Bradley Jones
Employee: Bradley Jones
Contractor: Hill Batfield
```

分析：首先来看一下该程序清单所做的修改。为减少代码，删除了一些构造函数。更重要的是，第20行是做了重要修改的第一个地方，Person类的displayFullName方法被声明为虚拟的，这指出：如果赋给Person变量的是派生类对象，则调用该方法时，实际将调用派生类中相应的方法

第 39 行的 Employee 类是从 Person 类派生而来的，这里包含第二个重要的修改——使用的是关键字 override，而不是 new。这表明：使用值为 Employee 对象的 Person 变量调用 displayFullName 方法时，调用的将是 Employee 类中的该方法。

为使该程序清单更有趣一些，并帮助说明，第 46～63 行加入了另一个从 Person 类派生而来的类 Contractors，它有自己的数据成员，用于存储雇佣的咨询人员所属的公司。这个类也包含一个 displayFullName 方法的覆盖版本。被调用时，该方法指出这个人是承包者（contractor）。

NameApp 类中的 Main 方法被修改得更为简单明了。第 70～72 行声明了三个 Person 变量，但每一个都被赋给不同的对象。第 70 行赋给的是一个 Person 对象；第 71 行是 Employee 对象；而第 72 行是 Contractor 对象。

警告：虽然第70～72行的每个变量（Brad、me和Greg）被赋给了不同类型的对象，但只能使用它们被声明的类型Person内的数据成员和方法。

第 74～76 行使用虚拟方法和覆盖方法。虽然用于调用 displayFullName 方法的三个变量都是 Person 类型的，但调用的是它们被实际赋给的值所属类中相应的方法，而并不都调用 Person 类的 displayFullName 方法。您几乎总是希望如此。

11.4 抽象类

在程序清单 11.4 中，并不一定非得在声明 Employee 和 Contractor 类的 diaplayFullName 方法时，使用关键字 override。如果将第 39 和 59 行修改为使用 new 关键字，而不是 override：

```
public new void displayFullName( )
```

则程序清单的运行结果将不同，如下所示：

```
Bradley Jones
Bradley Jones
Hill Batfield
```

为何会出现这种情况呢？虽然基类的 displayFullName 被声明为虚拟的（以便实现多态），但要根据赋给变量的数据类型来调用相应的方法，还必须在派生类的方法中使用关键字 override。

可以强制派生类覆盖基类的方法：将基类的方法声明为抽象的。为此，需要使用关键字 abstract。抽象方法没有方法体，由派生类来提供。

当方法被声明为抽象的时，其所属的类也必须被声明为抽象的。程序清单 11.5 使用了抽象类，其中也包含 Person、Employee 和 Contractor 类。

程序清单 11.5　inherit05.cs：使用抽象类

```
1: // inherit05.cs - Abstract Methods
2: //=============================================
3: using System;
4:
```

```
 5: abstract class Person
 6: {
 7:    protected string firstName;
 8:    protected string lastName;
 9:
10:    public Person()
11:    {
12:    }
13:
14:    public Person(string fn, string ln)
15:    {
16:       firstName = fn;
17:       lastName = ln;
18:    }
19:
20:    public abstract void displayFullName();
21: }
22:
23: class Employee : Person
24: {
25:    public ushort hireYear;
26:
27:    public Employee() : base()
28:    {
29:    }
30:
31:    public Employee(string fn, string ln, ushort hy) : base(fn, ln)
32:    {
33:       hireYear = hy;
34:    }
35:
36:    public override void displayFullName()
37:    {
38:       Console.WriteLine("Employee: {0} {1}", firstName, lastName);
39:    }
40: }
41:
42: // A new class derived from Person...
43: class Contractor : Person
44: {
45:    public string company;
46:
47:    public Contractor() : base()
48:    {
49:    }
50:
51:    public Contractor(string fn, string ln, string c) : base(fn, ln)
52:    {
53:       company = c;
54:    }
```

```
55:
56:    public override void displayFullName()
57:    {
58:       Console.WriteLine("Contractor: {0} {1}", firstName, lastName);
59:    }
60: }
61:
62: class NameApp
63: {
64:    public static void Main()
65:    {
66:
67: //       Person Brad = new Person("Bradley", "Jones");
68:       Person me   = new Employee("Bradley", "Jones", 1983);
69:       Person Greg = new Contractor("Hill", "Batfield", "Data Diggers");
70:
71: //       Brad.displayFullName();
72:       me.displayFullName();
73:       Greg.displayFullName();
74:    }
75: }
```

该程序清单的输出如下：

```
Employee: Bradley Jones
Contractor: Hill Batfield
```

分析： 特别要注意的是该程序清单的第20行，dispalyFullName方法被声明为抽象的。这表明该方法将在派生类中实现，因此这里没有该方法的方法体。

注意： 第20行是以分号结束的。

由于 Person 类有一个抽象方法，因此类本身也必须被声明为抽象的。所以，第 5 行加入了关键字 abstract。

在第 36 和 56 行的 Employee 和 Contractor 类都实现了覆盖方法 displayFullName。最后，应用程序类中的第 68 和 69 行将 Employee 和 Contractor 对象赋给了两个 Person 变量。当第 72 和 73 行调用 displayFullName 方法时，实际执行的是变量包含的数据类型所属类的方法，而不是其被声明的数据类型所属的方法。

第 67 和 71 行被注释掉，以防止它们执行。如果将第 67 行的注释标记删除，试图创建一个 Person 对象，将发生如下错误：

```
Inherit05.cs(67,22): error CS0144: Cannot create an instance of the abstract class or
interface 'person'
```

也可以尝试将第 36 和 56 行的关键字 override 改为 new，如果基类相应的方法是抽象的，则这将导致如下错误：

```
Inherit05.cs(23,7): error CS0534: 'Employee' does not implement inherited abstract member
'Person.displayFullName()'
Inherit05.cs(20,25): (Location of symbol related to previous error)
Inherit05.cs(43,7): error CS0534: 'Contractor' does not implement inherited abstract member
```

```
'Person.displayFullName()'
Inherit05.cs(20,25): (Location of symbol related to previous error)
```

编译器将确保基类的抽象方法被正确地覆盖。

11.5 密封类

创建抽象类时，期望从它派生出其他类。如果要禁止类被继承，该如何办呢？如果要封锁类，又该如何办呢？

C#提供了关键字 sealed 来防止类被继承。在定义类时包含限定符 sealed，可以禁止它被继承。程序清单 11.6 演示了一个非常简单的被密封（sealed）的类。

程序清单 11.6 inherit06.cs：密封类

```
// inherit06.cs - Sealed Classes
//=========================================
using System;

sealed public class number
{
   private float pi;

   public number()
   {
     pi = 3.14159F;
   }

   public float PI
   {
     get {
       return pi;
     }
   }
}

//public class numbers : number
//{
//  public float myVal = 123.456F;
//}

class myApp
{
   public static void Main()
   {
     number myNumbers = new number();
     Console.WriteLine("PI = {0}", myNumbers.PI);

//     numbers moreNumbers = new numbers();
//     Console.WriteLine("PI = {0}", moreNumbers.PI);
```

```
36: //     Console.WriteLine("myVal = {0}", moreNumbers.myVal);
37:    }
38: }keyword to
```

该程序清单的输出如下：

```
PI = 3.14159
```

该程序清单的大部分代码都简单易懂。第 5 行声明 number 类时使用了限定符 sealed。如果删除第 22～25 行的注释标记，并重新编译该程序清单，将出现以下错误：

```
inherit06b.cs(22,14): error CS0509: 'numbers': cannot inherit from sealed class 'number'
inherit06b.cs(5,14): (Location of symbol related to previous error)
```

这是由于密封类不能被继承，第 22 行试图继承 number 类，但没有成功。

注意：如果将密封类的数据成员声明为protected，编译器将发出警告。应该将数据成员声明为私有的，因为其所属的类不会被继承。

11.6 终极基类：Object

C#中的一切都是类，C#中的终极基类是 Object，它是.NET 框架类层次结构中的根类。这意味着它是最初的基类。

基于今天介绍的知识，可以说 C#中的任何东西都是一个 Object，也就是说所有的数据类型和类都是从 Object 类派生而来的，同时所有的.NET 类中都包含 Object 类中的所有方法。

11.6.1 Object 类中的方法

Object 实例有两个有趣的方法，因此所有的类都有这两个方法。它们是 GetType 和 ToString。前者返回对象的数据类型；后者返回一个表示当前对象的字符串。程序清单 11.7 通过前面创建的 PI 类调用了这些方法。

程序清单 11.7 obj.cs：任何东西都是 Object

```
 1: // obj.cs - Object Properties
 2: //=============================================
 3: using System;
 4:
 5: sealed class PI
 6: {
 7:    public static float nbr;
 8:
 9:    static PI()
10:    {
11:       nbr = 3.14159F;
12:    }
13:
14:    static public float val()
15:    {
16:       return(nbr);
17:    }
```

```
18: }
19:
20: class myApp
21: {
22:   public static void Main()
23:   {
24:     Console.WriteLine("PI = {0}", PI.val());
25:
26:     Object x = new PI();
27:     Console.WriteLine("ToString: {0}", x.ToString());
28:     Console.WriteLine("Type: {0}", x.GetType());
29:
30:     Console.WriteLine("ToString: {0}", 123.ToString());
31:     Console.WriteLine("Type: {0}", 123.GetType());
32:   }
33: }
```

该程序清单的输出如下：

```
PI = 3.14159
ToString: PI
Type: PI
ToString: 123
Type: System.Int32
```

分析：出于简化的目的，该程序清单的第一部分使用了程序清单11.6中的密封类PI，不同的是myApp类。第24行使用PI类的val方法来显示PI类中静态数据成员的值。从输出可以知道，这仍然是3.14159。但第26～31行的内容是新的。

第 26 行声明了一个名为 x 的变量，其数据类型为 Object，但指向的却是一个 PI 对象。由于 Object 类是所有类（包含该程序清单中创建的 PI 类）的基类，因此可以使用 Object 变量指向一个新的 PI 对象。第 27 和 28 行调用了 Object 类的两个方法：GetType 和 ToString。这些方法指出，x 的类型是 PI，它存储的是一个 PI 对象。

第 30 和 31 行包含一些奇怪的东西。别忘了，在 C#中，所有的东西（包括字面值）都是基于类的，而所有的类都是从 Object 派生而来的。这意味着诸如数字 123 等字面值实际上也是对象。使用从 Object 类派生而来的所有对象都可用的方法，将数字 123 转换为字符串的结果是 123（字符串）。另外，数字 123 的数据类型是 System.Int32（这是与 C#中的 int 对应的.NET 框架数据类型）。

11.6.2 装箱和拆箱

更深入地了解派生对象之间的关系后，我们来讨论另一个主题——装箱（boxing）和拆箱（unboxing）。

前面指出过，C#中的一切都是对象。这不太准确，而是所有的东西都可以被看作对象。前几天介绍过，值类型和引用类型的存储方式是不同的，而对象是引用类型。然而，程序清单 11.7 却像处理对象一样来处理字面值，这是如何实现的？

在 C#中，可以将值类型转换为对象，这能自动进行。在程序清单 11.7 中，值 123 被隐式地转换为一个对象。别忘了，所有的对象都是从终极基类 Object 派生而来的。对于 Object 对象以及从 Object 派生而来的对象，可以执行的操作很多，其中包含确定其类型。

新术语： *装箱*（boxing）指的是将值类型转换为引用类型（对象）；*拆箱*指的是显式地将引用类型转换为值类型。被拆箱的值必须存储到相应的数据类型变量中。

拆箱需要显式地将对象转换为值，这可以通过强制转换来实现。程序清单 11.8 演示了简单的装箱和拆箱操作。图 11.2 和图 11.3 帮助说明该程序清单完成的工作。

程序清单 11.8 boxit.cs：装箱和拆箱

```
1: // boxIt.cs - boxing and unboxing
2: //=============================================
3: using System;
4:
5: class myApp
6: {
7:    public static void Main()
8:    {
9:
10:      float val = 3.14F;      // Assign a value type a value
11:      object boxed = val;     // boxing val into boxed
12:
13:      float unboxed = (float) boxed;  // unboxing boxed into unboxed
14:
15:      Console.WriteLine("val: {0}", val);
16:      Console.WriteLine("boxed: {0}", boxed);
17:      Console.WriteLine("unboxed: {0}", unboxed);
18:
19:      Console.WriteLine("\nTypes...");
20:      Console.WriteLine("val: {0}", val.GetType());
21:      Console.WriteLine("boxed: {0}", boxed.GetType());
22:      Console.WriteLine("unboxed: {0}", unboxed.GetType());
23:   }
24: }
```

该程序清单的输出如下：

```
val: 3.14
boxed: 3.14
unboxed: 3.14

Types...
val: System.Single
boxed: System.Single
unboxed: System.Single
```

分析： 该程序清单重点介绍装箱和拆箱。第10行声明了一个值数据类型变量，并将值3.14赋给它。第11行执行装箱，将值类型val装箱到变量boxed中，后者是一个object变量。图11.2示出了val和boxed的存储方式，以说明它们之间的差别。

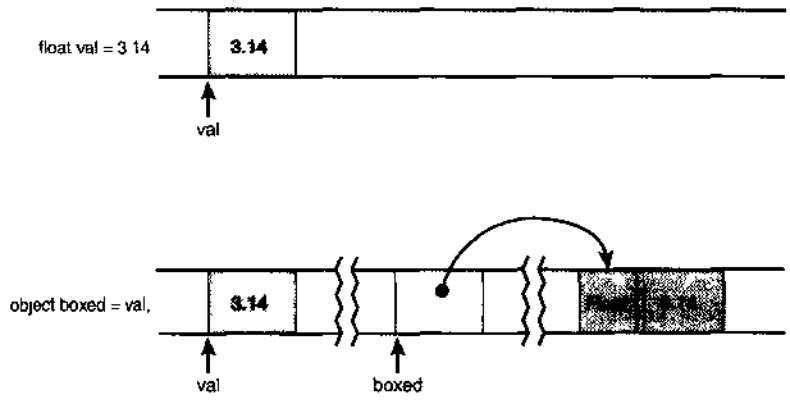


图 11.2 将值装箱

第 13 行将 boxed 中的值拆箱，并将其存储到变量 unboxed 中，后者是一个值类型变量，存储的是 boxed 中的值的拷贝——3.14。图 11.3 说明了这两个变量在内存中是如何存储的。

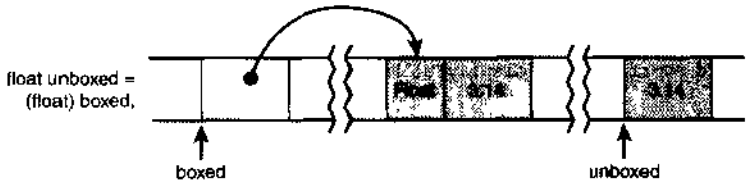


图 11.3 将值拆箱

第 20～22 行分别通过这三个变量调用 Object 的方法。您知道，val 和 unboxed 都是值类型，因此您可能会问，由于值类型并不真正存储其类型，而只存储其值，GetType 方法如何起作用呢？第 20 和 22 行，值类型自动装箱，因而转换为对象。这样，便可以调用诸如 GetType 等方法了。

11.7 将关键字 is 和 as 用于类——类转换

有两个关键字可用于类：is 和 as。

11.7.1 关键字 is

关键字 is 用于确定变量是否为指定的类型。is 语句的格式如下：

```
(expression is type)
```

其中 expression 的结果为引用类型，type 是一种有效的类型，通常是类。

如果 expression 与 type 兼容，则返回 true；否则返回 false。程序清单 11.9 虽然不实用，但说明了 is 的价值。

程序清单 11.9 islist.cs：使用关键字 is

```
1: // islist.cs - Using the is keyword
2: //=========================================
3: using System;
4:
5: class Person
```

```
 6: {
 7:    protected string Name;
 8:
 9:    public Person()  { }
10:
11:    public Person(string n)  { Name = n; }
12:
13:    public virtual void displayFullName()
14:    {
15:       Console.WriteLine("Name: {0}", Name);
16:    }
17: }
18:
19: class Employee : Person
20: {
21:    public Employee() : base()  { }
22:
23:    public Employee(string n) : base(n)  { }
24:
25:    public override void displayFullName()
26:    {
27:       Console.WriteLine("Employee: {0}", Name);
28:    }
29: }
30:
31: class IsApp
32: {
33:    public static void Main()
34:    {
35:       Person pers = new Person();
36:       Object emp = new Employee();
37:       string str = "String";
38:
39:       if( pers is Person )
40:          Console.WriteLine("pers is a Person");
41:       else
42:          Console.WriteLine("pers is NOT a Person");
43:
44:       if( pers is Object )
45:          Console.WriteLine("pers is an Object");
46:       else
47:          Console.WriteLine("pers is NOT an Object");
48:
49:       if( pers is Employee )
50:          Console.WriteLine("pers is an Employee");
51:       else
52:          Console.WriteLine("pers is NOT an Employee");
53:
54:       if( emp is Person )
55:          Console.WriteLine("emp is a Person");
```

```
56:       else
57:          Console.WriteLine("emp is NOT a Person");
58:
59:       if( str is Person )
60:          Console.WriteLine("str is a Person");
61:       else
62:          Console.WriteLine("str is NOT a Person");
63:    }
64: }
```

该程序清单的输出如下：

```
pers is a Person
pers is an Object
pers is NOT an Employee
emp is a Person
str is NOT a Person
```

分析：编译该程序清单时，编译器可能提出警告。这是因为对编译器而言，其中的一些is比较是显而易见的，因此它告诉您它们总是正确的。对于在程序运行时检查引用变量的类型而言，关键字is是一个非常棒的工具。

该程序清单首先声明了两个类，然后是 Main()方法。这些类的功能很少。第一个类是 Person；第二个类是 Employee，它是从 Person 派生而来的。然后，Main()方法使用了这些类。

第 35～37 行声明了三个变量。第一个是 Person 变量 pers，指向一个 Person 对象；第二个是 Object 变量 emp，指向一个 Employee 对象。您知道，可以将任何类型的对象赋给其基类变量，这意味着可以将 Employee 对象赋给 Object 变量、Person 变量或 Employee 变量。第 37 行声明了一个字符串变量。

其余的代码执行简单的检查，看这三个变量的类型是否不同。您可以再看一下输出，确定哪些比较返回 true，哪些返回 false。

注意：变量pers是Person和Object；变量emp是Employee、Person和Object。如果您不这么认为，则应该复习今天的课程。

11.7.2 关键字 as

as 运算符的功能与强制转换类似，将对象强制转换为另一种类型。目标类型必须与源类型兼容。as 的使用格式如下：

```
expression as DataType
```

其中 expression 的结果为一个引用类型，DataType 也是一种引用类型。相应的强制转换格式如下：

```
(DataType) expression
```

虽然 as 关键字的功能与强制转换类似，但它们并不是同一回事。使用强制转换时，如果出现问题——比如试图强制将字符串转换为数字，将引发异常。

但使用 as 关键字时，如果源类型无法转换为目标类型，则首先将其值设置为 null，然后再转换为目标类型，因此不会引发异常。

11.8 由不同类型的对象组成的数组

在今天课程的最后，我们来讨论另一个主题。通过使用关键字 as 和 is，您将有更大的能力。通过使用基类类型来定义数组变量，可以创建包含不同类型对象的数组。程序清单 11.10 演示了如何使用关键字 as 以及在同一个数组中存储不同类型的对象。这些不同的数据类型必须位于同一个继承层次结构中。

程序清单 11.10 objs.cs：对象数组

```
// objs.cs - Using an array containing different types
//=============================================
using System;

public class Person
{
   public string Name;

   public Person()
   {
   }

   public Person(string nm)
   {
      Name = nm;
   }

   public virtual void displayFullName()
   {
      Console.WriteLine("Person: {0}", Name);
   }
}

class Employee : Person
{
//   public ushort hireYear;

   public Employee() : base()
   {
   }

   public Employee(string nm) : base(nm)
   {
   }

   public override void displayFullName()
   {
      Console.WriteLine("Employee: {0}", Name);
   }
}
```

```
41:
42: // A new class derived from Person...
43: class Contractor : Person
44: {
45: //  public string company;
46:
47:    public Contractor() : base()
48:    {
49:    }
50:
51:    public Contractor(string nm) : base(nm)
52:    {
53:    }
54:
55:    public override void displayFullName()
56:    {
57:       Console.WriteLine("Contractor: {0}", Name);
58:    }
59: }
60:
61: class NameApp
62: {
63:    public static void Main()
64:    {
65:       Person [] myCompany = new Person[5];
66:       int ctr = 0;
67:       string buffer;
68:
69:       do
70:       {
71:          do
72:          {
73:             Console.Write("\nEnter \'c\' for Contractor, \'e\' for Employee then press
ENTER: ");
74:             buffer = Console.ReadLine();
75:          } while (buffer == "");
76:
77:          if ( buffer[0] == 'c' || buffer[0] == 'C' )
78:          {
79:             Console.Write("\nEnter the contractor\'s name: ");
80:             buffer = Console.ReadLine();
81:             // do other Contractor stuff...
82:             Contractor contr = new Contractor(buffer);
83:             myCompany[ctr] = contr as Person;
84:          }
85:          else
86:          if ( buffer[0] == 'e' || buffer[0] == 'E' )
87:          {
88:             Console.Write("\nEnter the employee\'s name: ");
89:             buffer = Console.ReadLine();
```

```
90:             // Do other employee stuff...
91:             Employee emp = new Employee(buffer);
92:             myCompany[ctr] = emp as Person;
93:           }
94:           else
95:           {
96:             Person pers = new Person("Not an Employee or Contractor");
97:             myCompany[ctr] = pers;
98:           }
99:
100:           ctr++;
101:
102:       } while ( ctr < 5 );
103:
104:       // Display the results of what was entered....
105:
106:       Console.WriteLine( "\n\n\n==========================");
107:
108:       for( ctr = 0; ctr < 5; ctr++ )
109:       {
110:         if( myCompany[ctr] is Employee )
111:         {
112:           Console.WriteLine("Employee: {0}", myCompany[ctr].Name);
113:         }
114:         else
115:         if( myCompany[ctr] is Contractor )
116:         {
117:          Console.WriteLine("Contractor: {0}", myCompany[ctr].Name);
118:         }
119:         else
120:         {
121:           Console.WriteLine("Person: {0}", myCompany[ctr].Name);
122:         }
123:       }
124:       Console.WriteLine( "==========================");
125:     }
126: }
```

该程序清单的输入如下：

```
Enter 'c' for Contractor, 'e' for Employee then press ENTER: C

Enter the contractor's name: Conner Bradshaw

Enter 'c' for Contractor, 'e' for Employee then press ENTER: E

Enter the employee's name: Tyler Truman

Enter 'c' for Contractor, 'e' for Employee then press ENTER: e
```

```
Enter the employee's name: Cody McCracker

Enter 'c' for Contractor, 'e' for Employee then press ENTER: c

Enter the contractor's name: Johnny Appleseed

Enter 'c' for Contractor, 'e' for Employee then press ENTER: z


==========================
Contractor: Conner Bradshaw
Employee: Tyler Truman
Employee: Cody McCracker
Contractor: Johnny Appleseed
Person: Not an Employee or Contractor
==========================
```

分析：与以前的程序清单相比，该程序清单很长，但却只实现一部分功能。今天的一个练习将要求您对其进行扩展。

该程序清单旨在让您能够将输入人员名单。这里只能输入5个，但您可以让用户不断输入，直到输入某个值为止。程序提示用户输入e或c，来表示这个人是雇员还是承包者。然后，根据用户的输入再用相应的提示，让用户输入姓名。您可以要求用户提供其他的信息，但这里没有这样做。

如果用户输入的不是c或e，则程序将姓名设置为一条错误信息。您很可能使用不同的逻辑。您应该注意到了，虽然程序提示用户输入c或e，但如果输入的是C或E，则仍然管用。

您应该熟悉该程序清单的大部分代码。其中定义的类已被最大限度的压缩。第26和45行被注释掉了，今天的一个练习将要求您使用这些数据成员。

从第63行开始是该应用程序的Main方法。第69～102行包含一个do...while语句，它让用户输入人名。第71～75行包含一个嵌套do...while语句，它提示用户输入e或c，以指示人的类型。第74行使用ReadLine来获取用户的回答。如果用户按Enter键，将再次进行提示。

用户输入值后，if...else语句被用来决定执行什么操作。第77行只检查用户输入的文本中的第一个字符，该字符被存储在字符串数组的第一个元素（buffer[0]）中。

如果用户输入的第一个字符为c，则执行第79～84行。第79行要求用户输入承包者的姓名，这里使用的是Write方法，而不是WriteLine，这样用户便可以在提示所在的行输入姓名。如果使用WriteLine，则将进行换行，用户将在接下来的一行中输入姓名。

第80行使用ReadLine方法读取用户输入的姓名。第82行创建一个名为contr的Contractor对象，并使用ReadLine读取的姓名初始化它。第83行将该新对象赋给数组myCompany。由于myCompany是一个Person数组，因此contr变量将作为Person类型被赋给该数组。由于Person是Contractor的基类，因此可以这样做——今天课程的前面介绍过了。

第108～123行遍历myCompany数组，将其每个元素中的姓名打印到屏幕上。第110行检查数组元素是否为Employee，如果是，则显示对应于雇员的输出。如果不是，则第115行的if语句检查它是不是Constrator，如果是，则打印一条消息。如果不是，则打印一条消息，指出它只是一个Person。

第124行打印虚线，以格式化输出。然后程序结束。

使用is和as让您能够在同一个数组中存储数据类型不同的对象，只要这些对象的基类相同即

可。由于所有的对象都是从 Object 派生而来的，因此总能找到相同的基类，从而实现上述目的。该程序清单说明了面向对象编程的一些重要特性。

11.9 总结

这是到目前为止内容最多的课程，也是最重要的。今天介绍了继承。您学习了如何创建基类和继承它们。另外，您还学习了将影响您如何使用基类和派生类的不同关键字（如 abstract、virtual 和 protected）以及如何密封类以禁止它被继承。

然后，您学习了如何使用这样的对象，即其包含的值与被声明的不同。可以将对象赋给其基类变量，并访问该变量。另外，可以使用 as 关键字将对象强制转换为其他类型，as 关键字的功能与强制转换运算类似，只是如果转换时出现问题，将把被转换的变量设置为 null，而不是引发异常。另外，还可以使用 is 关键字来确定对象的类型。

11.10 问与答

问：可以继承使用其他语言编写的类吗？

答：可以。.NET 的特性之一便是，类可以从其他语言编写的类继承而来。这意味着 C#类可以从使用其他语言编写的类派生而来。另外，其他语言的程序员也可以将 C#类用作基类。

问：程序清单 11.3 包含一个将派生类对象赋给基类变量的例子，那么可以将基类对象赋给派生类变量吗？

答：可以。只要仔细，便可以。在程序清单 11.3 中，将派生类对象赋给了一个基类变量。虽然只能使用那些通过基类能够建立的东西，但派生类的其他部分并没有丢失。如果基类变量的值为派生类对象，则可以将该基类变量赋给派生类变量。这种赋值是通过强制转换实现的。对于程序清单 11.3，如果在第 55 行后面加上下面的代码，将不会出错：

```
Employee you = (Employee) Brad;
```

这是合法的，因为 Brad 的值是一个 Employee 对象。如果 Brad 的值不是 Employee 对象，则上述代码将引发非法强制转换异常（System.InvalidCastException）。

问：何为数据或方法覆盖？

答：数据或方法覆盖指的是在派生类中创建的方法或数据元素将替换基类的方法或数据元素，此时需要使用关键字 new。

问：何为强制向上转换（upcasting）和强制向下转换（downcasting）？

答：强制向下转换指的是将对象强制转换为其派生类对象；强制向上转换指的是强制将对象转换为其基类对象。在 C#中，强制向上转换是安全的，是一种隐式操作；而强制向下转换是不安全的，要强制向下转换必须显式进行转换。

11.11 作 业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

11.11.1 小测验

1. 在 C#中，可以使用多少个类来派生出一个新类?

2. 下述哪个术语的含义与基类相同?

a. 父类。

b. 派生类。

c. 子类。

3. 哪种存取限定符用于禁止在类的外面访问数据? 哪种存取限定符只允许数据在其所属的类及其派生类中使用?

4. 如何覆盖基类的方法?

5. 可以在基类中使用哪个关键字来确保派生类创建自己的方法版本?

6. 哪个关键字用于禁止类被继承?

7. 指出两个所有的类都有的方法。

8. 哪个类是终极基类，所有的类都是从它派生而来的?

9. 装箱有何功能?

10. 关键字 as 有何用途?

11.11.2 练习

1. 编写为 ABC 类声明构造函数的方法头，它接受两个 int 参数 ARG1 和 ARG2。该构造函数调用基类的构造函数，并将 ARG2 传递给它。调用是在方法头中完成的：

```
public ABC (int ARG1, int ARG2): base (ARG2)
{
}
```

2. 修改下面的类，以禁止它被用作基类：

```
 1: class aLetter
 2: {
 3:     private static char A_ch;
 4:
 5:     public char ch
 6:     {
 7:       get { return A_ch; }
 8:       set { A_ch = value; }
 9:     }
10:
11:     static aLetter()
12:     {
13:       A_ch = 'X';
14:     }
15: }
```

3. 下面的代码有问题，错误在哪一行?

```
// Bug Buster

// Class definition for Person would need to be included here...

class NameApp
```

```
{
    public static void Main()
    {
      Person me = new Person();
      Object you = new Object();

      me = you;

      System.Console.WriteLine("Type: {0}", me.GetType());
    }
}
```

4. 选做题：对程序清单 11.9 进行修改，设置 hireyear 或 company 的值。并在适当的位置打印这些值。

第 12 天课程
输入和输出

前几天介绍了很多 C#开发的核心主题，这些主题对于开发专业级的应用程序至关重要。介绍其他核心主题之前，今天先换换口味，介绍以下内容：

- 输出和输出之间的差别；
- 更多用于在控制台中显示信息的格式化选项；
- 从控制台读取信息；
- 格式化和使用字符串；
- 流的概念；
- 操纵基本的文件信息。

注意：与本书中其他的课程相比，今天的课程包含多得多的参考信息。

12.1　理解控制台输入和输出

前面介绍过术语输入和输出。今天回过头来重点介绍以更好的格式提供输出。另外，还将更详细地介绍如何通过控制台输入从用户那里获取的信息。

另外，今天还将更详细地介绍字符串。Write 和 WriteLine 方法实际上在幕后对字符串进行格式化。今天您还将学习使用其他方法格式化字符串。

12.2　格式化信息

显示信息时，如果首先将其转换为字符串，则显示起来最为简单。您知道，Console 类的 Write 和 WriteLine 方法使用字符串来显示输出。另外，.NET 框架提供了大量可用于处理字符串的方法和说明符（specifier）。说明符指示要格式的信息。

格式说明符可用于任何字符串。接下来的几节将介绍大量的格式说明符，这包括用于以下方面的说明符：

- 标准数字格式；
- 格式化金额；
- 格式化指数数值；
- 格式化指数；

- 创建自定义数字格式；
- 格式化日期和时间；
- 格式化枚举。

使用这些格式说明符的方式很多。最显而易见的方式是在 Write 和 WriteLine 方法中使用说明符来提供额外的格式编排。

有些情况下，可以在调用 ToString 方法时使用格式说明符。前一天介绍过，Object 类包含一个 ToString 方法。由于所有的类都是从 Object 派生而来的，因此所有的对象都可以使用 ToString 方法。对于诸如 Int32 等类，可以给该方法传递格式化说明符，以格式化数据。例如，如果 var 是一个 int 变量，包含的值为 123，则通过下面的代码行使用金额格式标识符（"C"）返回值$123.00：

```
var.ToString("C");
```

第三种使用说明符的方式是通过 string 数据类型。string 类包含一个名为 Format 的静态方法。由于该方法是静态的，因此可以以 string.Format 的方式直接使用它。该方法的使用格式与 Console 中显示文本的方法相同：

```
string newString = string.Fromat("format_string", value(2) );
```

其中 newString 是格式化后得到的字符串；format_string 是一个包含格式化说明符的字符串，可用的说明符与 Write 和 WriteLine 方法相同；value(s)包含要被格式化到字符串中的值。

注意：在介绍说明符时，将介绍更多的关于格式化值的知识。

字符 C 是用于金额的格式说明符。前面说过，可以将其放在引号内，并作为参数传递给 ToString 方法，来指定这种格式。当格式化字符串中的信息时，如使用 WriteLine，您将该格式说明符与变量占位符放在一起。其基本格式如下：

```
{hldr:X#}
```

其中 hldr 是变量的占位符编号；X 是用于格式化数字的说明符，对于金额格式为 C；#是一个可选值，是需要的位数。根据说明符，该数字可执行不同类型的填充。对于金额说明符，该数字指定小数位数。程序清单 12.1 演示了前面介绍的三种使用说明符的方式，它包含一个使用金额说明符和其他说明符的小例子。今天课程的后面将更详细地介绍这些说明符以及其他说明符。

程序清单 12.1　formatit.cs：基本的格式化方式

```
 1: // formatit.cs - Different places for specifiers
 2: //            to be used.
 3: //-------------------------------------------------
 4:
 5: using System;
 6:
 7: class myApp
 8: {
 9:   public static void Main()
10:   {
11:     int var = 12345;
12:
13:     // Format using WriteLine
14:
```

```
15:      Console.Write("You can format text using Write");
16:      Console.WriteLine(" and WriteLine. You can insert");
17:      Console.Write("variables (such as {0}) into a string", var );
18:      Console.WriteLine(" as well as do other formatting!");
19:      Console.WriteLine("\n{0:C}\n{0:C4}", var);
20:      Console.WriteLine("\n{0:f}\n{0:f3}", var);
21:
22:
23:      // Format using ToString
24:
25:      string str1 = var.ToString("C");
26:      string str2 = var.ToString("C3");
27:      string str3 = var.ToString("E8");
28:
29:      Console.WriteLine("\nYou can also format using ToString");
30:      Console.WriteLine(str1);
31:      Console.WriteLine(str2);
32:      Console.WriteLine(str3);
33:
34:      // Formatting with string.Format
35:
36:      string str4 = string.Format("\nOr, you can use string.Format: ");
37:      string str5 = string.Format("Nbr {0:F3} \n{0:C} \n{0:C0}", var);
38:
39:      Console.WriteLine(str4);
40:      Console.WriteLine(str5);
41:   }
42: }
```

该程序清单的输出如下：

```
You can format text using Write and WriteLine. You can insert
variables (such as 12345) into a string as well as do other formatting!

$12,345.00
$12,345.0000

12345.00
12345.000

You can also format using ToString
$12,345.00
$12,345.000
1.23450000E+004
Or, you can use string.Format:
Nbr 12345.000
$12,345.00
$12,345
```

分析：这里不打算对该程序清单做完整的分析。这里提供该程序清单，只是为了简略地介绍一下可进行的格式化。该程序清单完成了一些格式化操作。var中的数字被格式化为金额、小数和指数表示法。这里要提到的更重要的一点是，位于说明符后面的数字决定包含的小数位数。

12.2.1 格式化数字

可用于格式化数值的格式说明符很多，表 12.1 列出了这些说明符。

表 12.1　　用于格式化数值的字符

说 明 符	描　述	默认格式	输出范例
C或c	金额	$xx,xxx.xx（$xx,xxx.xx）	$12,345.67（$12,345.67）
D或d	十进制数	xxxxxxx	1234567
		-xxxxxxx	-1234567
E或e	指数表示法	x.xxxxxxE+xxx	1.234567E+123
		x.xxxxxxe+xxx	1.234567e+123
		-x.xxxxxxE+xxx	-1.234567E+123
		-x.xxxxxxe+xxx	-1.234567e+123
		x.xxxxxxE-xxx	1.234567E-123
		x.xxxxxxe-xxx	1.234567e-123
		-x.xxxxxxE-xxx	-1.234567E-123
		-x.xxxxxxe-xxx	-1.234567e-123
F或f	定点数	xxxxxx.xx	1234567.89
		-xxxxxx.xx	-1234567.89
N或n	数值	xx,xxx.xx	12,345.67
		-xx,xxx.xx	-12,345.67
X或x	十六进制数		12d687
			12D687
G或g	通用	随具体情况而异（将使用最紧凑的格式）	
R或r	来回转换（Roundtrip）	将数字转换为字符串后，再转换为数字时，精度不变	

可以按前面介绍的方式使用上述格式说明符以及今天介绍的其他格式说明符。在 ToString 方法中使用说明符时，将相应的字符放在引号内，并将其作为参数传递。前面将 123 格式化为金额的范例中：

```
string newString = var.ToStirng("C");
```

newString 的值将为$123.00。接下来的各节将简要地讨论上述各种格式。

警告：格式说明符可能随系统的区域设置而异。

12.2.1.1　标准格式（定点、十进制和数值）

标准格式说明符用于相应的数值类型。F 用于浮点数，D 用于标准的整数（如整型和长整型）。如果试图将 D 用于浮点数，将出现异常。

说明符 N 以更好的格式表示数字，它在数字格式中加入两位小数和逗号。

12.2.1.2　格式化金额

您知道，金额说明符是 C。可以单独使用 C，使金额带两位小数。如果希望不带小数，可以使用 C0。0 指示不要包含小数部分。

12.2.1.3　格式化指数数字

由于指数数字非常大或非常小，因此使用科学计数法来表示。说明符 E 和 e 用于格式化这种数字。输出中 E 的大小写与格式化说明符相同。

12.2.1.4　数字的通用格式

通用格式说明符（G 或 g）用于将数字格式化为最紧凑的字符串格式。这种格式符根据具体的数字决定指数表示法更紧凑还是标准格式更紧凑，并返回更紧凑的一种格式。程序清单 12.2 说明了这种说明符可能返回的不同格式。

程序清单 12.2　general.cs：使用通用说明符

```
 1: // general.cs - Using the General format specifier
 2: //-----------------------------------------------
 3: using System;
 4:
 5: class myApp
 6: {
 7:   public static void Main()
 8:   {
 9:     float f1 = .000000789F;
10:     float f2 = 1.2F;
11:
12:     Console.WriteLine("f1  ({0:f}). Format (G): {0:G}", f1);
13:     Console.WriteLine("f2  ({0:f}). Format (G): {0:G}", f2);
14:   }
15: }
```

该程序清单的输出如下：

```
f1  (0.00). Format (G): 7.89E-7
f2  (1.20). Format (G): 1.2
```

分析：该程序清单初始化并打印两个变量。第9和10行创建了两个变量，一个是非常小的小数，另一个是简单的数字。

第 12 和 13 行将这些值写入到控制台。其中每一行的第一个占位符使用说明符 F 将浮点数显示为定点数；第二个占位符使用格式符 G 以通用格式打印相应的变量。从输出中可以知道，将 f1 以指数方法表示更简明，而 f2 以常规格式表示更简明。

12.2.1.5 格式化十六进制数

十六进制数是以 16 为基数的计数法的，这种计数法通常用于计算机中，附录 D 介绍了如何使用十六进制。字母 x（大写或小写）用于将数字表示为十六进制。十六进制说明符自动将值转换为十六进制。

12.2.1.6 保持精度（来回转换）

将数字从一种格式转换为另一种格式时，可能会降低精度。说明符 R（或 r）用于将字符串转换为数字时保持其精度。使用该说明符后，运行阶段环境将尽可能保持原有数字的精度。

12.2.1.7 使用模式定义（picture definition）创建自定义格式

有时候，需要更精确地控制数字的格式。例如，格式化驾驶证号或社会保险号时，可能想加入短划线；而对于电话号码，则想加入括号和短划线。表 12.2 列出了一些可结合使用说明符来自定义输出格式的格式化字符。程序清单 12.3 包含这些说明符的范例。

表 12.2 用于模式定义的格式说明符

说 明 符	描 述
0	零占位符，如果可能，则填充位
#	空占位符，如果可能，则填充位
.	显示一个句点，用作小数点
,	使用逗号将数字分组，也可用作倍增器（multiplier），见程序清单 12.3
%	将数字显示为百分比值（例如，1.00 被显示为 100%）
\	用于指示要打印的特殊字符。这可以是转移字符之一，如换行字符（\n）
'xyz'	显示单引号内的文本
"xyz"	显示双引号内的文本

程序清单 12.3 picts.cs：使用模式（picture）说明符

```
// picts.cs - Using picture specifiers
//-----------------------------------------------

using System;

class myApp
{
  public static void Main()
  {
     int var1 = 1234;
     float var2 = 12.34F;

     // Zero formatter
     Console.WriteLine("\nZero...");
     Console.WriteLine("{0} -->{0:0000000}", var1);
     Console.WriteLine("{0} -->{0:0000000}", var2);

```

```
18:      // Space formatter
19:      Console.WriteLine("\nSpace...");
20:      Console.WriteLine("{0} -->{0:0####}<--", var1);
21:      Console.WriteLine("{0} -->{0:0####}<--", var2);
22:
23:      // Group separator and mulitplier (,)
24:      Console.WriteLine("\nGroup Multiplier...");
25:      Console.WriteLine("{0} -->{0:0,,}<--", 1000000);
26:      Console.WriteLine("Group Separator...");
27:      Console.WriteLine("{0} -->{0:##,###,##0}<--", 2000000);
28:      Console.WriteLine("{0} -->{0:##,###,##0}<--", 3);
29:
30:      // Percentage formatter
31:      Console.WriteLine("\nPercentage...");
32:      Console.WriteLine("{0} -->{0:0%}<--", var1);
33:      Console.WriteLine("{0} -->{0:0%}<--", var2);
34:
35:     // Literal formatting
36:      Console.WriteLine("\nLiteral Formatting...");
37:      Console.WriteLine("{0} -->{0:'My Number: '0}<--", var1);
38:      Console.WriteLine("{0} -->{0:'My Number: '0}<--", var2);
39:      Console.WriteLine("\n{0} -->{0:Mine: 0}<--", var1);
40:      Console.WriteLine("{0} -->{0:Mine: 0}<--", var2);
41:   }
42: }
```

该程序清单的输出如下：

```
Zero...
1234 -->0001234
12.34 -->0000012

Space...
1234 -->01234<--
12.34 -->00012<--

Group Multiplier...
1000000 -->1<--
Group Separator...
2000000 -->2,000,000<--
3 -->3<--

Percentage...
1234 -->123400%<--
12.34 -->1234%<--

Literal Formatting...
1234 -->My Number: 1234<--
12.34 -->My Number: 12<--
```

```
1234 -->Mine: 1234<--
12.34 -->Mine: 12<--
```

分析：该程序清单将表12.2列出的格式说明符用于两个变量。从注释和输出可以知道，这些说明符是如何起作用的。

12.2.1.8 格式化负数

有时候，需要以不同于正数的方式来处理负数。前面介绍的说明符同时适用于正数和负数。

可以将指定格式的占位符分为两部分或三部分。如果分为两部分，则前一部分用于正数和零，而后一部分用于负数。如果分为三部分，则第一部分用于正数，中间部分用于零，最后一部分用于负数。

使用分号将占位符分成多个部分，其中每部分都包含占位符编号。例如，对于正数，精度为3位，对于负数，精度为5位，对于0精度为1位，则可以使用下面的格式：

```
{0:D3;D5;'0'}
```

程序清单12.4包含这样的范例。

程序清单12.4 threeways.cs

```
// threeway.cs - Controlling the formatting of numbers
//---------------------------------------------------

using System;

class myApp
{
  public static void Main()
  {
    Console.WriteLine("\nExample 1...");
    for ( int x = -100; x <= 100; x += 100 )
    {
      Console.WriteLine("{0:000;-00000;'0'}", x);
    }

    Console.WriteLine("\nExample 2...");
    for ( int x = -100; x <= 100; x += 100 )
    {
      Console.WriteLine("{0:Pos: 0;Neg: -0;Zero}", x);
    }

    Console.WriteLine("\nExample 3...");
    for ( int x = -100; x <= 100; x += 100 )
    {
      Console.WriteLine("{0:You Win!;You Lose!;You Broke Even!}", x);
    }
  }
}
```

该程序清单的输出如下：

```
Example 1...
-00100
0
100

Example 2...
Neg: -100
Zero
Pos: 100

Example 3...
You Lose!
You Broke Even!
You Win!
```

分析：该程序清单演示了如何将自定义格式分为三部分。它使用for循环来创建了一个负数，然后将其递增为0和正数。对于这三个值，可以使用同一个WriteLine来显示它们。这种操作执行了3次，完成三个不同的例子。

从第 13 行可以知道，对于正数，打印结果中将至少包含三位，因为第一部分包含三个 0；对于负数，则打印结果中将包含负号，后面至少跟五个数字，这是因为格式中包含表示负号的短划线和 5 个零。如果值为零，则打印的将是 0。

在第二个和第三个例子的格式中包含文本。它们之间的差别在于，第二个例子中还包含占位符 0，以便打印实际的值；而第三个例子则没有，只打印文本。

从这三个例子可以知道，很容易根据变量的正负，提供不同的格式。

12.2.2　格式化日期和时间

日期和时间也可以被格式化。有很多说明符可用于格式化从星期几到完整的日期和时间字符串等。介绍这些格式字符之前，先来介绍如何获取日期和时间。

12.2.2.1　获取日期和时间

C#和.NET 框架提供了一个用于存储日期和时间的类——DateTime，它位于名称空间 System 中。DateTime 类存储完整的日期和时间。

DateTime 类有许多很有用的属性和方法，另外还有一些静态成员。您最有可能使用的两个静态属性是 Now 和 Today。Now 包含它被使用时的日期和时间，而 Today 返回当前的日期。由于它们是静态属性，因此可以通过类名（而不是实例名称）来取得它们的值：

```
DateTime.Now
DateTime.Today
```

有关所有方法和属性的信息，请参阅在线文档。下面是一些很有用的方法和属性：

- Date：返回 DateTime 对象的日期部分；
- Month：返回 DateTime 对象的月份部分；
- Day：返回 DateTime 对象的天部分；
- Year：返回 DateTime 对象的年份部分；
- DayOfWeek：返回 DateTime 对象的星期部分（即星期几）；
- DayOfYear：指出 DateTime 对象中日期属于该年份的多少天；

- TimeOfDay：返回 DateTime 对象的时间部分；
- Hour：返回 DateTime 对象的小时部分；
- Minute：返回 DateTime 对象的分钟部分；
- Second：返回 DateTime 对象的秒部分；
- Millisecond：返回 DateTime 对象的毫秒部分；
- Ticks：返回 DateTime 对象对应的 100 纳秒数。

注意：DateTime.Today值提供当前的日期，而不包含时间。

12.2.2.2 格式化日期和时间

用于日期、时间或两者的说明符很多。这包括以长格式或短格式显示信息。表 12.3 列出了这些日期和时间说明符。

表 12.3 日期和时间格式字符

说 明 符	描 述	默认格式	输出范例
d	短日期	mm/dd/yyyy	5/6/2001
D	长日期	day, month dd, yyyy	Sunday, May 06, 2001
f	完整的日期/短时间	day, month dd, yyyy hh:mm AM/PM	Sunday, May 06, 2001 12:30 PM
F	完整的日期/完整的时间	day, month dd, yyyy HH:mm:ss AM/PM	Sunday, May 06, 2001 12:30:54 PM
g	短日期/短时间	mm/dd/yyyy HH:mm	6/5/2001 12:30 PM
G	短日期/长时间	mm/dd/yyyy hh:mm:ss	6/5/2001 12:30:54 PM
M 或 m	月份和天	month dd	May 06
R 或 r	RFC1123	ddd, dd Month yyyy hh:mm:ss GMT	Sun, 06 May 2001 12:30:54 GMT
s	合适的（sortable）	yyyy-mm-dd hh:mm:ss	2001-05-06T12:30:54
t	短时间	hh:mm AM/PM	12:30 PM
T	长时间	hh:mm:ss AM/PM	12:30:54 PM
u	合适的（通用的）	yyyy-mm-dd hh:mm:ss	2001-05-06 12:30:54Z
U	合适的（通用的）	Day, month dd, yyy hh:mm:ss AM/PM	Sunday, May 06, 2001 12:30:54 PM
Y 或 y	年份/月份	month, yyyy	May, 2001

警告：s是一种用于打印合适日期的说明符，注意它是小写的。大写的S不是有效的说明符，使用它将引发异常。

日期和时间说明符使用起来很容易。程序清单 12.5 定义了一个日期变量，并以表 12.3 列出的各种格式打印该变量。

程序清单 12.5　dtformat.cs：日期格式

```
 1: // dtformat.cs - date/time formats
 2: //---------------------------------------------
 3:
 4: using System;
 5:
 6: class myApp
 7: {
 8:   public static void Main()
 9:   {
10:     DateTime CurrTime = DateTime.Now;
11:
12:     Console.WriteLine("d: {0:d}", CurrTime );
13:     Console.WriteLine("D: {0:D}", CurrTime );
14:     Console.WriteLine("f: {0:f}", CurrTime );
15:     Console.WriteLine("F: {0:F}", CurrTime );
16:     Console.WriteLine("g: {0:g}", CurrTime );
17:     Console.WriteLine("G: {0:G}", CurrTime );
18:     Console.WriteLine("m: {0:m}", CurrTime );
19:     Console.WriteLine("M: {0:M}", CurrTime );
20:     Console.WriteLine("r: {0:r}", CurrTime );
21:     Console.WriteLine("R: {0:R}", CurrTime );
22:     Console.WriteLine("s: {0:s}", CurrTime );
23: //    Console.WriteLine("S: {0:S}", CurrTime );  // error!!!
24:     Console.WriteLine("t: {0:t}", CurrTime );
25:     Console.WriteLine("T: {0:T}", CurrTime );
26:     Console.WriteLine("u: {0:u}", CurrTime );
27:     Console.WriteLine("U: {0:U}", CurrTime );
28:     Console.WriteLine("y: {0:y}", CurrTime );
29:     Console.WriteLine("Y: {0:Y}", CurrTime );
30:   }
31: }
```

该程序清单的输出如下：

```
d: 5/6/2001
D: Sunday, May 06, 2001
f: Sunday, May 06, 2001 1:06 PM
F: Sunday, May 06, 2001 1:06:51 PM
g: 5/6/2001 1:06 PM
G: 5/6/2001 1:06:51 PM
m: May 06
M: May 06
r: Sun, 06 May 2001 13:06:51 GMT
```

```
R: Sun, 06 May 2001 13:06:51 GMT
s: 2001-05-06T13:06:51
t: 1:06 PM
T: 1:06:51 PM
u: 2001-05-06 13:06:51Z
U: Sunday, May 06, 2001 6:06:51 PM
y: May, 2001
Y: May, 2001
```

分析：第10行声明了一个用于存储日期和时间的对象，这是使用DateTime类实现的。该对象名为CurrentTime，并将DateTime类的静态成员Now的值赋给它。Now提供当前的日期和时间。从输出可以知道，作者是在五月份运行该程序清单的。第12～29行以各种格式打印该日期和时间。

第 23 行被注释掉，它使用了非法的说明符 S。如果将该行的注释标记删除，该程序清单将引发异常。

12.2.3 显示枚举中的值

在第 8 天学习枚举时，您发现，使用 Write 和 WriteLine 输出枚举时，显示的是枚举的描述性值，而不是数字型值。通过格式化字符串，可以控制输出是数字型值还是文本值，还可以将其强制显示为十六进制。表 12.4 列出了用于枚举的格式化字符，而程序清单 12.6 则使用了这些字符。

表 12.4　　用于枚举的格式化字符

说 明 符	描　述
D或d	显示枚举元素的数值型值
G或g	显示枚举元素的字符串值
X或x	以十六进制格式显示枚举元素的数字型值

程序清单 12.6　enums.cs：格式化枚举值

```
 1: // enums.cs - enumerator formats
 2: //---------------------------------------------
 3:
 4: using System;
 5:
 6: class myApp
 7: {
 8:   enum Pet
 9:   {
10:       Cat,
11:       Dog,
12:       Fish,
13:       Snake,
14:       Rat,
15:       Hamster,
16:       Bird
17:   }
18:
```

```
19:   public static void Main()
20:   {
21:      Pet myPet = Pet.Fish;
22:      Pet yourPet = Pet.Hamster;
23:
24:      Console.WriteLine("Using myPet: ");
25:      Console.WriteLine("d: {0:d}", myPet );
26:      Console.WriteLine("D: {0:D}", myPet );
27:      Console.WriteLine("g: {0:g}", myPet );
28:      Console.WriteLine("G: {0:G}", myPet );
29:      Console.WriteLine("x: {0:x}", myPet );
30:      Console.WriteLine("X: {0:X}", myPet );
31:
32:      Console.WriteLine("\nUsing yourPet: ");
33:      Console.WriteLine("d: {0:d}", yourPet );
34:      Console.WriteLine("D: {0:D}", yourPet );
35:      Console.WriteLine("g: {0:g}", yourPet );
36:      Console.WriteLine("G: {0:G}", yourPet );
37:      Console.WriteLine("x: {0:x}", yourPet );
38:      Console.WriteLine("X: {0:X}", yourPet );
39:   }
40: }
```

该程序清单的输出如下：

```
Using myPet:
d: 2
D: 2
g: Fish
G: Fish
x: 00000002
X: 00000002

Using yourPet:
d: 5
D: 5
g: Hamster
G: Hamster
x: 00000005
X: 00000005
```

分析：该程序清单声明了一个用于存储宠物的枚举。第21和22行创建了两个枚举对象myPet和yourPet，用于说明格式说明符。第24～30行将说明符用于对象myPet；而第32～38则用于yourPet。从输出可以知道，说明符的大小写无关紧要。

12.3　使用字符串

介绍了关于格式化字符串的所有知识后，我们回过头来更详细地介绍一些字符串方面的知识。

字符串是一种特殊的数据类型，可以存储文本信息。

您知道，string 是一个 C#关键字，它指的是 System 名称空间中的 String 类，因此字符串具有 String 类的所有属性和方法。

存储在字符串中的值是不能修改的。当您调用字符串方法或修改字符串时，实际上将创建一个新的字符串。如果试图修改字符串中的字符，将发生错误。程序清单 12.7 是一个简单的例子，它证明了这一点。

程序清单 12.7 str_errocs：字符串不能被修改

```
1: // str_err.cs - Bad listing. Generates error
2: //-------------------------------------------------
3: using System;
4:
5: class myApp
6: {
7:   public static void Main()
8:   {
9:     string str1 = "abcdefghijklmnop";
10:
11:    str1[5] = 'X';      // ERROR!!!
12:
13:    Console.WriteLine( str1 );
14:   }
15: }
```

编译该程序清单时，将出现如下所示的错误：

```
str_err.cs(11,6): error CS0200: Property or indexer 'string.this[int]' cannot be assigned
to - it is read only
```

分析：该程序清单证明了您不能修改字符串。字符串是一个由字符组成的数组。第6行试图将第6个字符改为大写的字母X。这将导致错误，因为您不能修改字符串的值。

您可能认为，如果字符串不能修改，则其用途将很有限。另外您可能会问，如何将方法和属性用于字符串。您将发现，修改字符串的方法实际上将创建一个新的字符串。另外，如果您确实需要修改字符串，则可以使用 C#提供的另一个类：StringBuilder，这将在今天课程的后面介绍。

注意：字符串被认为是永远不变的，这意味着不能修改。

12.3.1 字符串方法

有很多非常有用的方法可用于字符串，表 12.5 列出了一些重要的字符串方法，并对其用途进行了描述。

表 12.5 常用的字符串方法

方 法	描 述
静 态 方 法	
Compare	比较两个字符串的值

续表

方　法	描　述
静 态 方 法	
CompareOrdinal	比较两个字符串的值，但不根据语言或其他国际化问题进行校正
Concat	将两个或更多的字符串合并成一个
Copy	使用已有的字符串创建一个新的字符串
Equals	比较两个字符串，以确定它们是否包含相同的值。如果是，则返回 true；否则返回 false
Foramt	使用相应的字符串值替换格式说明符，说明符在今天课程的前面介绍过了
Join	将两个或更多的字符串合并为一个，在原来的字符串之间加上指定的“分隔符字符串”
每个实例都有的方法和属性	
Char	返回指定位置处的字符
Clone	返回存储的字符串的拷贝
CompareTo	将当前字符串同另一个字符串进行比较。如果当前字符串更小，则返回一个负数；如果等于，则返回 0；如果更大，则返回一个正数
CopyTo	将字符串的一部分或全部复制到一个新的字符串或字符数组中
EndsWith	确定字符串是否以某个字符或字符串结尾，如果是则返回 true，否则返回 false
Equals	比较两个字符串，以确定它们是否包含相同的值。如果是，则返回 true；否则返回 false
IndexOf	返回字符串中第一次出现某个字符或字符串的索引（位置），如果没有这样的字符或字符串，则返回-1
Insert	将一个值插入到字符串中，这是通过返回一个新的字符串来实现的
LastIndexOf	返回字符串中最后一次出现某个字符或字符串的索引（位置），如果没有这样的字符或字符串，则返回-1
Length	返回字符串的长度，长度等于字符串包含的字符数
PadLeft	将字符串右对齐，并在左边填充指定的字符（或空格）
PadRight	将字符串左对齐，并在右边填充指定的字符（或空格）
Remove	从字符串的指定位置开始删除指定数目的字符
Split	执行的操作与 Join 相反，即根据指定的值，将字符串分成几个子串，指定的值被用作分拆点
StartsWith	确定字符串是否以某个字符串或字符串打头，如果是则返回 true，否则返回 false
Substring	从字符串中指定位置开始返回一个子串，也可以指定返回多少个字符，但不是必须的
ToCharArray	将当前字符串中的字符复制到一个 char 数组中
ToLower	返回当前字符串的小写

续表

方　法	描　述
每个实例都有的方法和属性	
ToUpper	返回当前字符串的大写
Trim	从当前字符串的开头和结尾删除指定的字符串
TrimEnd	从当前字符串的结尾删除指定的字符串
TrimStart	从当前字符串的开头删除指定的字符串

在本书余下的内容中，将使用其中的很多方法和属性。

12.3.2 特殊的字符串格式符——@

前面的程序清单中使用了很多特殊字符。例如，要在字符串中使用双引号，可以使用转移字符。下面的代码打印"Hello World"（包括双引号）：

```
System.Console.WriteLine("\"Hello World\"");
```

要在字符串中包含反斜杠，也必须使用转移字符：

```
System.Console.WriteLine("My Code: C:\\Books\\TyCSharp\\Originals\\");
```

C#提供了一个特殊的格式字符@，用于省略转移字符。当该字符位于字符串的前面时，字符串的值将被看作是字面值。事实上，这种字符串被称为"逐字字符串字面值"（verbatim string literal）。下面的语句与前面打印目录的语句等效：

```
System.Console.WriteLine(@"My Code: C:\Books\TYCSharp\Original");
```

使用字符串格式符@时，您将发现，唯一需要注意的地方是双引号。如果在使用@格式化的字符串中使用双引号，则需要使用两个双引号。下述代码与打印"Hello World"的范例等效：

```
System.Console.WriteLine(@"""Hello World""");
```

注意：您可能在想，不是说使用两对双引号吗，为何使用了三对双引号？这是因为其中的一对用于将字符串括起。在这个例子中，第一个双引号表示字符串的开始位置，但由于它后面跟了一个双引号，因此系统知道需要显示一个双引号。第四个双引号本来表示字符串的结束位置，但由于它后面还有双引号，因此被转换为显示一个双引号。第六个双引号检查是否结束字符串，由于它后面没有跟双引号，因此字符串至此结束。

12.3.3 创建字符串

名称空间 System.Text 中的 StringBuilder 类用于创建用来存储可修改字符串的对象。使用 StringBuilder 类创建的对象的工作与字符串类似，差别在于 StringBuilder 的方法可直接操纵存储的值。表 12.6 列出了 StringBuilder 对象的方法和属性。

表 12.6　　StringBuilder 的方法和属性

方法或属性	描　述
Append	将对象附加到当前 StringBuilder 的后面
AppendFormat	根据格式说明符将对象插入到字符串中
Capacity	设置或取得可以存储的字符数。Capacity 可以增加到等于 MaxCapacity
Chars	使用索引器表示法取得或设置给定索引位置的字符
EnsureCapacity	确保 StringBuilder 的容量至少大于指定的值。如果给 EnsureCapacity 传递了值，则 Capacity 属性将被设置为该值。如果 MaxCapacity 的值小于传递的值，将发生异常
Equals	确定当前 StringBuilder 是否等于传递的值
Insert	将对象放置到 StringBuilder 中的指定位置
Length	设置或取得当前存储在 StringBuilder 中的长度值。Length 不能大于 StringBuilder 的 Capacity。如果当前值小于 Length，则将截尾
MaxCapctity	取得 StringBuilder 的最大容量
Remove	从当前 StringBuilder 对象的指定位置开始，删除指定数目的字符。
Replace	将指定字符替换为新的字符
ToString	将 StringBuilder 转换为一个字符串

StringBuilder 类的使用方法与其他类相似。程序清单 12.8 使用了 StringBuilder 对象以及表 12.6 中的几个方法和属性。该程序清单让用户输入姓、名和中名，然后将它们组合成一个 StringBuilder 对象。

程序清单 12.8　stribylder.cs：使用 StringBuilder 类

```
// strbuilder.cs - String Builder
//---------------------------------------------
using System;
using System.Text;    //For StringBuilder

class buildName
{
  public static void Main()
  {
    StringBuilder name = new StringBuilder();
    string buffer;
    int marker = 0;

    Console.Write("\nEnter your first name: ");
    buffer = Console.ReadLine();

    if ( buffer != null )
    {
```

```
19:       name.Append(buffer);
20:       marker = name.Length;
21:     }
22:
23:     Console.Write("\nEnter your last name: ");
24:     buffer = Console.ReadLine();
25:
26:     if ( buffer != null )
27:     {
28:       name.Append(" ");
29:       name.Append(buffer);
30:     }
31:
32:     Console.Write("\nEnter your middle name: ");
33:     buffer = Console.ReadLine();
34:
35:     if ( buffer != null )
36:     {
37:       name.Insert(marker+1, buffer);
38:       name.Insert(marker+buffer.Length+1, " ");
39:     }
40:
41:     Console.WriteLine("\n\nFull name: {0}", name);
42:
43:     // Some stats....
44:     Console.WriteLine("\n\nInfo about StringBuilder string:");
45:     Console.WriteLine("value: {0}", name);
46:     Console.WriteLine("Capacity: {0}", name.Capacity);
47:     Console.WriteLine("Maximum Capacity: {0}", name.MaxCapacity);
48:     Console.WriteLine("Length: {0}", name.Length);
49:   }
50: }
```

该程序清单的输出如下：

```
Enter your first name: Bradley

Enter your last name: Jones

Enter your middle name: Lee

Full name: Bradley Lee Jones

Info about StringBuilder string:
value: Bradley Lee Jones
Capacity: 32
Maximum Capacity: 2147483647
Length: 17
```

分析：首先需要注意的是，第4行使用using语句将名称空间System.Text包含了进来。没有这条语句，将需要以全限定名称System.Text.StringBuilder来引用StringBuilder类。

第 10 行创建了一个名为 name 的 StringBuilder 对象，用于存储该程序清单将创建的字符串。第 11 行创建了一个名为 buffer 的字符串，用于取得用户输入的信息。这种信息是在第 15、24 和 33 行使用 Console 类的 ReadLine 方法来读取的。首先读取的是姓，它被追加到 StringBuilder 对象 name 中，由于该对象为空，因此姓被放在开始位置。姓的长度被赋给变量 marker，该变量被用来确定放置中名的位置。

接着取得的是名，它追加到 name 对象中（第 29 行），前面加入了一个空格（第 28 行）。最后，获得中名，并使用 Insert 方法将其插入到 name 对象的中间。前面保存的 marker 被用来确定插入中名的位置。

第 41 行显示得到的全名。第 44 ~ 48 行显示一些通用的信息。第 45 行打印了 StringBuilder 对象 name 的值；第 46 行打印 name 对象当前的容量；第 47 行打印最大容量；第 48 行打印 name 对象当前存储的值的长度。

12.4　从控制台获取信息

到目前为止，今天的课程主要介绍了如何格式化和显示信息。除了生成输出外，您还需要在获取输入（输入给程序的信息）方面有更灵活的方法。

本书前面使用过 Console 类的 Read 和 ReadLine 方法，接下来的几节将更详细地介绍如何获取输入并将其转换为更有用处的格式。

12.4.1　使用 Read 方法

System 名称空间中的 Console 类包含两个可用于从用户那里获取信息的方法：ReadLine 和 Read。

Read 方法每次从输入流中读取一个字符，并将该字符作为 int 值返回。如果读到了流的末尾，则返回-1。程序清单 12.9 使用 Read 方法来读取字符。

提示：要结束来自控制台的字符流，可以按Ctrl + z键。

程序清单 12.9　Read 方法

```
 1: // readit.cs - Read information from Console
 2: //---------------------------------------------------
 3: using System;
 4: using System.Text;
 5:
 6: class myApp
 7: {
 8:   public static void Main()
 9:   {
10:     StringBuilder Input = new StringBuilder();
11:
12:     int ival;
13:     char ch = ' ';
14:
15:     Console.WriteLine("Enter text. When done, press CTRL+Z:");
16:
```

```
17:     while ( true )
18:     {
19:       ival = Console.Read();
20:       if ( ival == - 1 )
21:         break;
22:       ch = ( char ) ival;
23:
24:       Input.Append(ch);
25:     }
26:     Console.WriteLine("\n\n==========>\n");
27:     Console.Write( Input );
28:     Console.Write("\n\n");
29:   }
30: }
```

该程序清单的输出如下：

```
Enter text. When done, press CTRL+Z:
Mary Had a little lamb,
Its fleece was white as snow.
Everywhere that Mary went,
that lamb was sure to go!
==========>
Mary Had a little lamb,
Its fleece was white as snow.
Everywhere that Mary went,
that lamb was sure to go!
```

分析：从输出可以知道，共输入了四行文本。输入完文本后，按Ctrl + Z来结束输入。

第 17～25 行使用了一个 While 循环来读取字符。实际的读取工作是在第 19 行进行的。读取的值被赋给变量 ival。如果 ival 等于-1，则表明到达了输入的末尾，因此使用 break 命令来退出循环。如果 ival 的值有效，则将其强制转换为字符值，如第 22 行所示。然后，将其追加到字符串 Input 中。

完成字符的输入后，第 27 行打印字符串 Input。所有的字符都被存储到 Input 中，包括回车和换行符。

注意：Windows和Web应用程序获取信息的方法与此不同。第15～17天将更详细地介绍在这些平台上如何获取信息。

12.4.2 使用 ReadLine 方法

本书前面已多次使用过 ReadLine 方法，您应该知道其功能。ReadLine 方法读取回车、换行符或两者之前的一行文本。如果达到流的末尾，则返回 null。

下面的程序清单与 12.9 类似。要结束该程序清单的运行，可按 Ctrl + z 键。

程序清单 12.10 readln.cs：使用 ReadLine 来读取字符

```
1: // readln.cs - Read information from Console
2: //---------------------------------------- -- ---
3: using System;
4: using System.Text;
5:
```

```
 6: class myApp
 7: {
 8:   public static void Main()
 9:   {
10:     StringBuilder Input = new StringBuilder();
11:     string buff;
12:
13:     Console.WriteLine("Enter text. When done, press Ctrl+Z:");
14:
15:     while ( (buff = Console.ReadLine()) != null )
16:     {
17:       Input.Append(buff);
18:       Input.Append("\n");
19:     }
20:     Console.WriteLine("\n\n==========>\n");
21:     Console.Write( Input );
22:     Console.Write("\n\n");
23:   }
24: }
```

该程序清单的输出如下：

```
Enter text. When done, press Ctrl+Z:
Twinkle, twinkle little star
How I wonder where you are
up above the sky so high
like a diamond in the sky

==========>

Twinkle, twinkle little star
How I wonder where you are
up above the sky so high
like a diamond in the sky
```

分析：该程序清单的功能与12.9相同，但它一次读取一行字符（第15行），而不是一个。如果读取的行为null，则输入结束。如果读取到了信息，则将其追加到Input字符串中（第17行）。由于读取的信息中不包括换行符，因此第18行在创建的字符串中加入了换行符，以便第21行的输出与输入的内容相同。

如果禁止使用 Ctrl + Z 呢？假设您希望输入一个空行来结束该程序。今天课程的练习 3 将要求您修改该程序清单，以便输入空行来结束程序。

12.4.3 Convert 类

使用 Read 和 ReadLine 方法读取信息的关键不是取得信息，而是将其转换为想要的格式。这可以是对字符串中的文本进行过滤，得到一个不同的字符串，或将其转换为一种不同的数据类型。

System 名称空间中有一个可用于将数据转换为其他数据类型的类：Convert 类。

Convert 类是一个被密封的类，包含大量的静态方法。这些方法可将数据转换为不同的数据类

型。由于这些方法是静态的，因此使用格式如下：

```
Convert.method( Orig_val);
```

这里假设您已经使用 using 语句将名称空间 System 包含进来了。其中，method 是要使用的转换方法的名称；Orig_val 是要转换为新类型的值。这个类位于基类库中，知道这一点非常重要。这意味着可在其他编程语言中使用这个类。Convert 类的方法将数据转换为基数据类型，而不是 C#数据类型。但不用着急，第 3 天的课程介绍过，每种 C#数据类型都有相应的基类型。

表 12.7 列出了 Convert 类的一些方法。完整的方法列表请参阅.NET 框架文档。程序清单 12.11 是一个使用 Convert 类方法的简短范例。该程序清单将 ReadLine 读取的字符串值转换为一个整数。

表 12.7　　转换数据类型的方法

方　法	目 标 类 型
ToBoolean	布尔型
ToByte	8 位的无符号整数
ToChar	Unicode 字符
ToDateTime	DateTime
ToDecimal	Decimal
ToDouble	双精度数
ToInt16	16 位的符号整数
ToInt32	32 位的符号整数
ToInt64	64 位的符号整数
ToSByte	8 位的符号整数
ToSingle	单精度浮点数
ToString	字符串
ToUInt16	16 位的无符号整数
ToUInt32	32 位的无符号整数
ToUInt64	64 位的无符号整数

程序清单 12.11　conv.cs：使用 Convert 的方法

```
1: // conv.cs - Converting to a data type
2: //---------------------------------------------
3: using System;
4: using System.Text;
5:
6: class myApp
```

```
 7: {
 8:   public static void Main()
 9:   {
10:     string buff;
11:     int age;
12:
13:     Console.Write("Enter your age: ");
14:
15:     buff = Console.ReadLine();
16:
17:     try
18:     {
19:       age = Convert.ToInt32(buff);
20:
21:       if( age < 21 )
22:         Console.WriteLine("You are under 21.");
23:       else
24:         Console.Write("You are 21 or older.");
25:     }
26:     catch( ArgumentException )
27:     {
28:       Console.WriteLine("No value was entered... (equal to null)");
29:     }
30:     catch( OverflowException )
31:     {
32:       Console.WriteLine("You entered a number that is too big or too small.");
33:     }
34:     catch( FormatException )
35:     {
36:       Console.WriteLine("You didn't enter a valid number.");
37:     }
38:     catch( Exception e )
39:     {
40:       Console.WriteLine("Something went wrong with the conversion.");
41:       throw;
42:     }
43:   }
44: }
```

下面是在命令行执行该程序清单多次得到的输出：

```
C:\Day12>conv
Enter your age: 12
You are under 21.

C:\Day12>conv
Enter your age: 21
You are 21 or older.

C: \Day12>conv
```

```
Enter your age: 65
You are 21 or older.

C:\Day12>conv
Enter your age: 9999999999999999
You entered a number that is too big or too small.

C:\\Day12>conv
Enter your age: abc
You didn't enter a valid number.

C:\Day12>conv
Enter your age: abc123
You didn't enter a valid number.

C:\Day12>conv
Enter your age: 123abc
You didn't enter a valid number.

C:\Day12>conv
Enter your age: 123 123
You didn't enter a valid number.
```

分析：您首先注意到的是，该程序清单使用了异常处理。大凡可能发生异常时，都应该使用异常处理。如果输入的信息不合适，则第19行使用的转换方法ToInt32将可能引发异常。第26、30和34行捕获三种不同的异常，第38行捕获预料之外的其他异常。如果没有出现异常，则根据年龄是否小于21岁，显示一条消息。

该程序清单让您知道如何从终端用户那里获取信息、将其转换为更有用的格式、并验证它以确保其合法。

12.5 总 结

今天的课程介绍了大量的内容。请给包含方法表的页做上标记，以方便您在后面编程时参考。

今天介绍了如何格式化信息，使之更好看。除了学习如何格式化常规数据类型外，您还学习了如何取得并格式化日期和时间。

今天课程的后半部分重点介绍了字符串以及可用于操纵它们的方法。由于字符串是永远不变的——不可修改，因此还介绍了 StringBuilder 类。通过这个类，您学习了如何操纵字符串信息。

最后，重点介绍了如何从控制台获取信息。您温习了 Read 和 ReadLine 方法，并将其与学到的格式化字符串和转换类 Convert 结合起来使用。现在，您知道如何从控制台读取信息，并将其转换为有用的格式（包括不同的数据类型）。

12.6 问与答

问：您说字符串不能被修改，而很多字符串方法看起来却修改了字符串，原因何在？

答：直接操纵字符串的方法并不修改原来的字符串，而是创建一个修改后的新字符串，并用它替换原来的字符串。对于 StringBuilder 类，则可以操纵原来的字符串。

问：您说 Convert 类处理的是基数据类型，我不明白您的意思，请解释。

答：您应该复习第 3 天的课程。当您编译程序时，C#数据类型将被转换为运行阶段环境中的数据类型。例如 C#类型 int 将被转换为 System.Int32。实际上，在程序中，int 和 System.Int32 可以互换

12.7 作　业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

12.7.1 小测验

1. 哪个方法可用于将数据类型 int 转换为字符串，并进行格式化？
2. string 类中的哪个方法可用于将信息格式化为一个新的字符串？
3. 哪个说明符可用于指示将数字格式化为金额？
4. 哪个说明符可用于指示将数字格式化为包含逗号和一位小数（如 123,890.5）？
5. 如果 x 的值为 123456789.876，则下述代码的输出是什么？

```
Console.WriteLine(" X is {0:'a value of  '#, .#'.'}", x);
```

6. 在下述情况下，应使用什么样的说明符：如果为正数，则将其显示为至少包含 5 位的 decimal 数；如果为负，则至少包含 8 位；如果为 0，则显示文本<empty>。
7. 如何取得当前的日期？
8. 字符串和 StringBuilder 对象之间的关键区别在哪里？
9. 哪个特殊字符用作字符串格式符，它对字符串有何影响？
10. 如何从字符串得到数字型值？

12.7.2 练习

1. 编写一行代码，将数字格式化为至少包含三位和两位小数。
2. 编写以“星期几，月份，日和四位年份”格式（如 Monday, January 1, 2002）打印日期值的代码。
3. 修改程序清单 12.10，以便程序在用户输入空行后结束。
4. 修改程序清单 12.11。如果用户输入的信息中包含空格，则对信息进行裁剪，只是用空格前面的部分，并对其进行转换。例如，如果用户输入 123 456，则只使用 123。
5. 选做题：编写一个这样的程序：让用户输入其全名、年龄和电话号码，以特定的格式显示这些信息，并显示用户的姓名的首字母。

第 13 天课程
接　　口

今天，将加深您对面向对象编程的理解，这包括通过使用接口来扩大继承和多态的用途。您将学习如何从多个来源继承特征以及如何对大量不同的数据类型执行方法。具体地说，今天将学习以下内容：

- 接口简介；
- 接口的基本结构；
- 定义和使用接口；
- 实现多个接口；
- 使用已有的接口派生出新的接口；
- 如何对类隐藏接口。

第 6 天介绍了关于类的知识，第 11 天介绍了如何使用一个类派生出另一个类。今天将介绍如何从多个来源继承特征，派生出新的类。

13.1 接　口

请看下面的类，这个类有何特点呢？

```
public abstract class cShape
{
    public abstract long Area();
    public abstract long Circumference();
    public abstract int sides();
}
```

上述类是一个抽象类，其所有的方法都是抽象的。抽象类已经在第 11 天介绍过了。

抽象类指的是至少包含一个抽象方法的类。而抽象方法指的是被继承时，必须覆盖的方法。

接口是另一种类似于类的引用类型，它与上面的 cShape 类极其类似。接口的用途是定义要声明的类中将包含什么，但不定义实际的功能。接口类似于抽象方法，与上面的类极其类似。

事实上，通过删除类和方法的限定符，并把关键字 class 改为 interface，可将 cShape 类改为接口：

```
public interface IShape
```

```
{
    long Area();
    long Circumference();
    int sides();
}
```

13.1.1 类和接口之比较

接口类似于纯粹的抽象类，它与类之间的差别很多。

首先（也是最主要的），接口不提供任何实现代码，这些代码是由实现接口提供的。接口提供关于将发生的情况的规范或指南，但不提供细节。

接口不同于类，其所有的成员都被视为公有的。如果试图为接口的成员声明一个不同的作用域限定符，将出错。

接口只包含方法、属性、事件和索引器（indexer），而不包含数据成员、构造函数和析构函数，也不能包含静态成员。

抽象类也具有上述特点，但在功能上不同于接口。阅读完今天的课程后，您便会理解这一点。

13.1.2 使用接口

接口的功能好像没有类那么强大，但它具备一些类无法实现的功能。类只能从另一个类派生而来，但类可以实现多个接口。另外，结构不能继承另一个结构或方法，但可以实现接口。后面将更详细地介绍如何实现多个接口。

注意：C#不像C++和其他面向对象语言那样支持多重继承。由于多重继承复杂的实现所带来的麻烦，多重继承被有意地删除了。C#通过允许实现多个接口提供了多重继承的功能和好处。

13.1.3 为何使用接口

使用接口有好处，从前几节可以知道一些。

首先，可以将接口作为一种给结构提供继承特性的途径。另外，可以在一个类中实现多个接口，从而获得抽象类无法获得的功能。

使用接口最大的价值之一是，可以给类添加通过其他方法无法实现的特征。如果将同样的特征添加到其他类中，便可以对其具备的功能做出假设。实际上，通过使用类，可以避免做这样的假设。

使用接口而不是类带来的另一个好处是，强制新的类实现接口定义的所有特征。如果继承带虚拟成员的基类，则可能不为虚拟成员提供实现代码。这使得新类和使用新类的程序可能出错。

13.2 定义接口

前面介绍过，接口是关于类需要实现的功能的指南，其基本结构如下：

```
interface IName
{
   members;
}
```

其中 IName 是接口的名称。可以为接口指定任何名称，但作者建议以 I 打头，以表明它是一个接口，这与大多数程序员的通常做法一致。